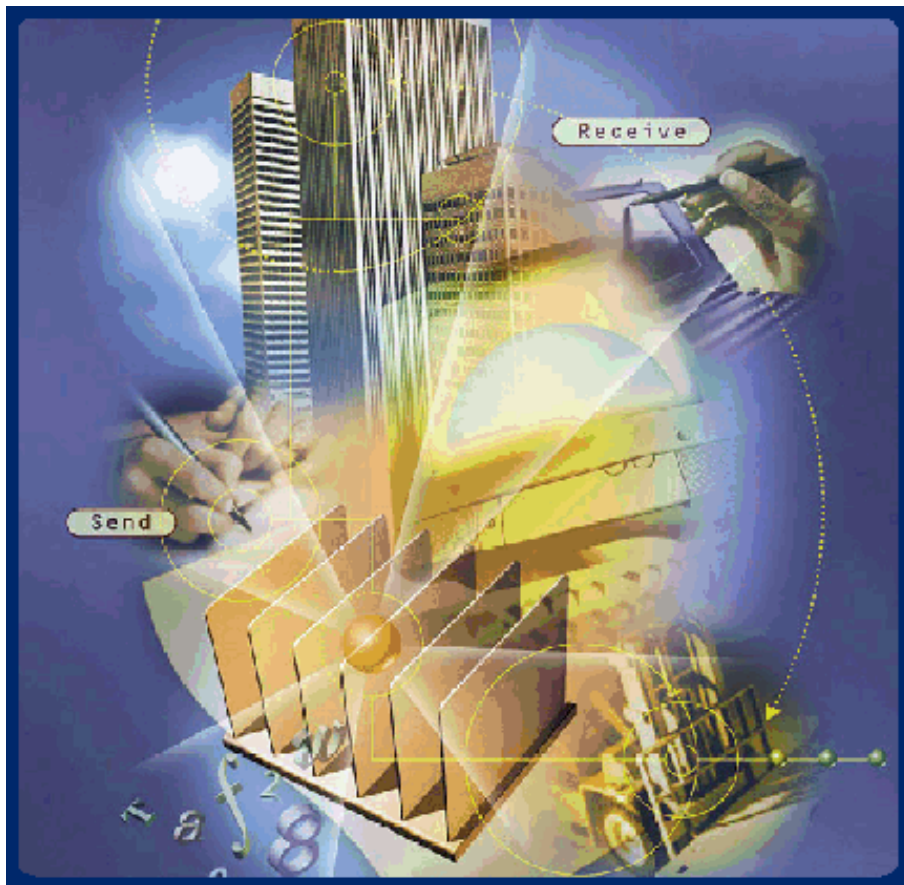


Progress®
SonicMQ™

**SonicMQ
Programming
Guide**



Copyright© 2000 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

Progress® is a registered trademark of Progress Software Corporation.

SonicMQ™, AppServer™, ProVision™, ProVision Plus™, Progress SmartObjects™, Apptivity™, and all other Progress product names are trademarks of Progress Software Corporation.

Progress SonicMQ™ contains the IBM® XML Parser for Java Edition and the IBM® Runtime Environment for Windows®, Java™ Technology Edition Version 1.1.8 Runtime Modules.© Copyright IBM Corporation 1998-1999. All rights reserved. U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

IBM® is a registered trademark of IBM Corporation. Java™ is a trademark of Sun Microsystems Inc. Windows® is a registered trademark of Microsoft Corp. All other company and product names are the trademarks or registered trademarks of their respective companies.



Printed in U.S.A.
November 2000

Contents

Preface	15
About This Manual	15
How This Book is Organized.....	16
Conventions in This Manual	17
Typographical Conventions and Syntax Notation.....	17
Note, Important, and Warning Flags.....	19
Available Documentation	20
Worldwide Technical Support.....	21
Chapter 1: Overview	23
About SonicMQ.....	23
Java Message Service	24
JMS: Key Component of the Java Platform for the Enterprise	24
JMS 1.0.2 Specification	24
Java Development Environment	25
Programming Concepts	25
Clients Connect to the SonicMQ Message Server Architecture	25
SonicMQ Is a JMS Provider	26
SonicMQ Messaging Models	27
SonicMQ Objects and Their Relationships.....	28
SonicMQ Object Model	29
Connections and Sessions.....	30
Producers and Consumers.....	31
Quality of Service and Protection	32
SonicMQ Clients	38
ActiveX/COM Client	38
Java Applet Client	38

Chapter 2: Examining the SonicMQ Samples	39
About SonicMQ Samples	39
SonicMQ Samples	40
Other Samples Available	41
Extending the Samples	42
How Security Impacts Client Activities	42
Running the SonicMQ Samples	43
Starting the Message Server Under Windows, Linux, or UNIX	43
Client Console Windows	44
Using the Sample Scripts	44
Using the SonicMQ Explorer	45
Chat and Talk Samples	46
Chat Application (Pub/Sub)	46
Talk Application (PTP)	47
Reviewing the Chat and Talk Samples	47
Samples of Additional Message Types	48
XML Messages	48
XML Messages (PTP)	49
XML Messages (Pub/Sub)	49
Map Messages (PTP)	49
Reviewing the Additional Message Type Samples	51
Message Traffic Monitor Samples	52
QueueMonitor Application (PTP)	53
MessageMonitor Application (Pub/Sub)	54
Transaction Samples	56
TransactedTalk Application (PTP)	56
TransactedChat Application (Pub/Sub)	57
Reviewing the Transaction Samples	58
Reliable, Persistent, and Durable Messaging Samples	60
Reliable Connections	60
ReliableTalk Application (PTP)	61
ReliableChat Application (Pub/Sub)	62
Persistent Storage Application (PTP)	63
DurableChat Application (Pub/Sub)	69
Reviewing Reliable, Persistent, and Durable Messaging	71
Request and Reply Samples	72
Request and Reply (PTP)	73
Request and Reply (Pub/Sub)	74
Reviewing the Request and Reply Samples	74

Selection and Wildcard Samples	75
SelectorTalk Application (PTP)	75
SelectorChat Application (Pub/Sub)	76
Hierarchical Chat Application (Pub/Sub)	77
Reviewing the Selection and Wildcard Samples	78
Test Loop Sample	78
QueueRoundTrip Application (PTP)	78
Extending the Samples	79
Use Common Topics Across Clients	79
Trying Different RoundTrip Settings	80
Modifying the MapMessage to Use Other Data Types	81
.	82
Modifying the XMLMessage to Show More Data	82
Using Samples with Security Initialized	85
Removing Security from the Database	88
Chapter 3: SonicMQ Client Sessions	89
About Client Sessions	89
Identifiers	89
ConnectID	89
User Name	90
ClientID	90
Subscription Name	91
Communication Layer	92
ConnectionFactory	93
Lookup a Stored Context	93
Direct Creation of the ConnectionFactory Object	95
Load Balancing and Failover Lists	97
Connection	98
Connection Retry	99
Session	100
Explicit Acknowledgement	100
Acknowledgement Mode	101
Transacted Sessions	102
Session Objects	103
create [Destination]	104
Point-to-Point: createQueue	104
Publish and Subscribe: createTopic	105
Using a Lookup for Destinations	105
Temporary Destinations	105

Contents

create [MessageProducer].	105
Point-to-Point: createSender	105
Publish and Subscribe: createPublisher	105
create [MessageConsumer].	106
Point-to-Point: createReceiver	106
Publish and Subscribe: create[Durable]Subscriber	106
create [Message].	106
Starting, Stopping, and Closing Connections	107
connect.start	107
connect.stop	107
Behavior of Producers and Consumers in a Stopped Connection	107
connect.close.	108
Behavior of Producers and Consumers in a Closed Connection	108
Closing a Session	108
Flow Control	109
Using Multiple Connections, Sessions, and Consumers.	111
Multiple Connections	111
Multiple Sessions on a Connection.	112
Coding Connections and Sessions	112
Get a Connection and Session	113
Using Active Pings to Monitor the Health of the Connection	114
Create Session Objects and the Listeners	114
Start the Connection	115
Handle Exceptions on the Connection	115
Handling Dropped Connection Errors Caught with Active Pings	116
Exception Listeners are Not Intended for JMS Errors	116
JMS Messaging Domains	117
Chapter 4: Messages	119
About Messages	119
Message Type	120
Creating a Message.	121
XML Type	121
Message Structure	122
Messages and Selectors	122
Message Header Fields	123
Setting Header Values When Sending/Publishing	126
Default Header Values	126

Message Properties	127
User-defined Properties	127
Provider-defined Properties (JMS_SonicMQ)	127
JMS-defined Properties (JMSX)	128
Setting Message Properties	129
Property Methods	129
propertyExists	130
clearProperties	130
set[type]Property	130
getPropertyNames	130
get[type]Property	131
Message Body	132
Setting the Message Body	132
Getting the Message Body	133
Getting the Body from an XML Type	133
Chapter 5: Message Producers and Consumers	135
About Message Producers and Message Consumers	135
Generic Messaging Model	135
Message Ordering and Reliability	136
Destinations	137
Steps in Message Production	138
Create the Topic Publisher on the PublisherSession Thread	138
Create the Producer on the Producer Session Thread	139
Create the Message Type and Set Its Body	139
Set Message Header Fields	139
Set the Message Properties	140
Produce the Message	140
Message Management by the Message Server	142
Message Listeners, Receivers, and Selectors	143
Message Listeners	143
Message Receiver	144
Receive	144
Receive with Timeout	144
Receive No Wait	144
Message Selector	145
Message Selector Syntax	146
Comparing Exact and Inexact Values	149

Contents

Steps in Listening, Receiving and Consuming Messages	150
Implement the Message Listener	150
Create the Destination and Consumer, then Listen	151
Handle a Received Message	151
Get Message Header Fields	152
Get Message Properties	152
Consume the message	153
Reply-to Mechanisms	153
Temporary Destinations Managed by a Requestor Helper Class	154
Requestor Application	154
Replier Application	154
Design for Handling Requests	155
Writing a Topic Requestor	155
Producers and Consumers in JMS Messaging Domains	157
Chapter 6: Point-to-Point Messaging	159
About Point-to-Point Messaging	159
Coding Queues, Senders, and Receivers	160
Coding Sample	160
Message Ordering and Reliability in PTP	161
Message Ordering	161
Reliability	161
Advantages and Constraints in PTP Domains	162
Multiple Receivers	163
Message Queue Listener	163
Message Queue Receiver	163
Prefetch Count and Threshold	165
Queue Browsing	166
createBrowser	166
createBrowserMessage (MessageSelector)	166
QueueBrowser Sample	167
Handling Undelivered Messages	168
Setting Important Messages to Get Saved If They Expire	168
Setting Quick Messages to Generate Administrative Notice	169
Life Cycle of a Guaranteed Message	170
Setting the Message to Be Preserved	170
Setting the Message to Generate an Administrative Event	170
Sending the Message	170

Letting the Message Get Delivered or Expire	170
Post-Processing of Expired Message	170
Programmer Callback for Undelivered Message Notification	172
Getting Messages Out of the Dead Message Queue	173
Chapter 7: Dynamic Routing Architecture	175
About Dynamic Routing	175
Message Behavior on Global and Local Queues	176
Undelivered Message Reason Codes	177
Sending to a Message Server Where Queues Exist	178
Sending to a Message Server Where Queues Do Not Exist	180
Sending to a Cluster Routing Node With Queues Everywhere	182
Send to a Cluster Routing Node With Queues in One Place	184
Reply-to Mechanisms for a DRA Application	186
Setting Applications to Use Simple Request Messages	186
Using Specific Shared Reply Queues	187
Chapter 8: Publish and Subscribe Messaging	189
About Publish and Subscribe Messaging	189
Coding Topics, Subscribers, Publishers, and Listeners	190
Topic	191
Publisher	192
Creating the Publisher	192
Creating the Message	193
Publishing to a Topic	193
Subscriber	195
Durable Subscriber	196
Durable Subscriptions Not Allowed for Temporary Topics	196
Unsubscribing from a Durable Subscription	196
Unsubscribing to Durable Subscription Requires Inactive Subscriber	197
Message Ordering and Reliability	198
General Services	198
Message Ordering	198
Reliability	199

Chapter 9: Hierarchical Name Spaces	201
About Hierarchical Name Spaces	201
Advantages of Hierarchical Name Spaces	201
Publishing a Message to a Topic	203
Topic Notation that Enables Topic Hierarchies	203
Reserved Characters when Publishing	203
Topic Structure, Syntax, and Semantics	203
Topic Syntax and Semantics	204
Message Server Management of Topic Hierarchies	204
Subscribing to Nodes in the Topic Hierarchy	205
Template Characters	205
Using Template Characters in Symmetric Hierarchies	207
Using Template Characters in Asymmetric Topic Hierarchies	208
Template Character for Subscribing to All Topics	209
Template Character for All Topics Under a Topic Hierarchy	209
Multiple Template Characters in an Expression	209
Examples of a Topic Name Space	210
Publishing Messages to a Hierarchical Topic	210
Subscribing to Sets of Hierarchical Topics	211
Chapter 10: Management API	213
About the Management API	213
Using the Management API	214
Samples that Use the Management API	215
Events	215
Accessing All Events	216
Accessing Selected Events	216
Piping Events Into a Log	217
Metrics	218
Piping Metrics Into a Log	219
Setup Queues	220
Show Setup	221
Accessing All Message Server Queue Information	221
Accessing Selected Message Server Queue Information	222
Shutdown	223
Chapter 11: Accessing SonicMQ Through ActiveX/COM Clients	225

About SonicMQ Through ActiveX/COM	225
Implementation Notes	226
Requirements for an ActiveX/COM Client	226
SonicMQ ActiveX/COM Sample	227
Visual Basic Code for the ActiveX/COM Sample	229
Tips and Techniques for SonicMQ ActiveX/COM	233
Identifiers	233
Handling Messages	235
XML Messages	235
Resource Management	235
Events	236
Connections	237
True ActiveX/COM Properties	237
Enumerations	238
Constants	239
Syntax for SonicMQ ActiveX/COM Method Names	239
Duplicate Names Are Differentiated	239
Java Method Overloading Is Handled	240
Interface Class Names Are Often Omitted	240
Interface Mappings	241
Connections and Sessions	242
Producers and Consumers	244
Publish and Subscribe (Topics)	246
Point-to-point (Queues)	249
Messages	252
Special Purpose	262
Chapter 12: Lookup of Administered Objects	265
About Administered Objects	265
Issues When Using Administered Objects	266
Creating New Administered Objects	266
Serialized Java Objects in a File System	267
Setting Up Serialized Objects	267
Using Serialized Objects	268
Using JNDI to Interface With a Directory Server	268
Index	271

List of Figures

Figure 1. Message Server Is a Hub for SonicMQ Client Applications	26
Figure 2. Client Application Using the SonicMQ JMS Provider	27
Figure 3. SonicMQ Object Relationships	28
Figure 4. JMS Object Model for the Point-to-Point Domain	29
Figure 5. Principal Interfaces for Point-to-Point	30
Figure 6. Principal Interfaces for Publish and Subscribe	30
Figure 7. Message Producers and Message Consumers	31
Figure 8. Using the Explorer to Maintain the Default Queues	45
Figure 9. QueueMonitor Window	53
Figure 10. Message Monitor Window	55
Figure 11. ReliableTalk Sample Trying to Reconnect	62
Figure 12. Sequence Diagram for the DurableChat Application	70
Figure 13. XMLMessage Parsed into a Document Object Model	84
Figure 14. XMLMessage as Tagged Text	85
Figure 15. Client - Message Server - Client Communications	92
Figure 16. Sessions in Connections from Connection Factories	92
Figure 17. ConnectionFactory Object Instantiated By Lookup of a Serialized Java Object	94
Figure 18. Alternate Connection Techniques Using Factory Objects or JNDI Lookup	94
Figure 19. Using a Constructor to Create a ConnectionFactory Object	95
Figure 20. Primary Session Objects	103
Figure 21. Types of SonicMQ Message Objects	104
Figure 22. Multiple Connections in a Client Application	111
Figure 23. Multiple Sessions on a Connection	112
Figure 24. SonicMQ Message Types	120
Figure 25. User-defined Properties	129
Figure 26. Generic Messaging Model	135
Figure 27. Session Objects in the JMS Domains	157
Figure 28. Point-to-Point Messaging Model	159
Figure 29. Message Server Where Specified Queues Exist	178
Figure 30. Message Server Where Specified Queues Do Not Exist	180
Figure 31. Cluster Routing Node Where Queues Exist On Every Server	182
Figure 32. Cluster Routing Node where Queues Exist on Only One Server	184
Figure 33. Publish and Subscribe Messaging Model	189
Figure 34. Explorer View of Creating a Publisher	192
Figure 35. Explorer View of a Message Header Fields After Publishing	194
Figure 36. Explorer View of Subscribing to a Topic	195

Figure 37. Topic Structure Without Hierarchies	202
Figure 38. Topic Structure With Hierarchies	202
Figure 39. Subscribing to the Topic Credit.U S A	205
Figure 40. Symmetric Topic Structure	207
Figure 41. Asymmetric Topic Structure	208
Figure 42. A Sample Hierarchy of Topics	210
Figure 43. Explorer View of a Newly Created Queue	220
Figure 44. SonicMQ ActiveX/COM Sample, <code>chat.frm</code> , in Visual Basic	227

List of Tables

Table 1. The SonicMQ Documentation Set	20
Table 2. Progress Software International Offices	22
Table 3. Services and Protection Available in SonicMQ Messaging	33
Table 4. Differences Between QueueMonitor and MessageMonitor	52
Table 5. Transacted Session Events by Message Role	102
Table 6. Connected Session Functionality Common to PTP and Pub/Sub	117
Table 7. Message Header Fields	123
Table 8. SonicMQ Provider-defined Properties	128
Table 9. JMSX Properties Used in SonicMQ	128
Table 10. Permitted Type Conversions for Message Properties	131
Table 11. How Message Producer Parameters Influence the Message Server	142
Table 12. Literal and Identifier Syntax in Message Selectors	146
Table 13. Operator and Expression Syntax in Message Selectors	148
Table 14. Comparison Test Syntax in Message Selectors	149
Table 15. Reply-To Mechanisms in Both Domains	153
Table 16. Messaging Subclasses in JMS Messaging	157
Table 17. Producer and Consumer Common to Both Messaging Models	158
Table 18. Advantages of the Point-to-Point Messaging Model	162
Table 19. Reason Codes for Undelivered Messages	177
Table 20. Routing Behavior on a Server Where Specified Queues Exist	179
Table 21. Routing Behavior on Server Where Specified Queues Do Not Exist	181
Table 22. Routing Behavior on a Cluster Node Where Queues Exist on Each Server	183
Table 23. Routing Behavior on Cluster Node Where Queues Exist on Only One Server	185
Table 24. True ActiveX/COM Properties in the SonicMQ ActiveX/COM Control	237
Table 25. Interface Mapping from SonicMQ to the ActiveX/COM Control	241

Contents

Table 26. Connection Interface	242
Table 27. Session Interface	243
Table 28. MessageConsumer Interface	244
Table 29. MessageListener Interface	245
Table 30. MessageProducer Interface	245
Table 31. DeliveryMode Interface	246
Table 32. TopicConnectionFactory Interface	246
Table 33. TopicConnection Interface	247
Table 34. TopicSession Interface	247
Table 35. Topic Interface (Extends Destination)	248
Table 36. TopicPublisher Interface	248
Table 37. TopicRequestor and TemporaryTopic (Extends Topic) Interfaces	248
Table 38. TopicSubscriber Interface	249
Table 39. QueueConnectionFactory Interface	249
Table 40. QueueConnection Interface	249
Table 41. QueueSession Interface	250
Table 42. Queue Interface (Extends Destination)	250
Table 43. QueueSender Interface (Extends MessageProducer)	251
Table 44. QueueRequestor and TemporaryQueue (Extends Queue) Interfaces	251
Table 45. QueueReceiver Interface (Extends MessageConsumer)	251
Table 46. QueueBrowser Interface	252
Table 47. Message Interface	252
Table 48. BytesMessage Interface (Extends Message)	256
Table 49. MapMessage Interface (Extends Message)	258
Table 50. StreamMessage Interface (Extends Message)	260
Table 51. TextMessage Interface (Extends Message)	261
Table 52. XMLMessage Interface (Extends TextMessage)	262
Table 53. Other Interfaces	262

Preface

This Preface covers the following topics:

- [“About This Manual”](#) describes this manual and its intended audience.
- [“Conventions in This Manual”](#) describes the text formatting, syntax notation, and flags used in this manual.
- [“Available Documentation”](#) describes the printed and online documentation that accompanies SonicMQ.
- [“Worldwide Technical Support”](#) provides information on contacting technical support.

About This Manual

Progress SonicMQ is a fast, flexible, scalable e-Business Message Server designed to simplify the development and integration of today's highly distributed applications and Internet-based business solutions. SonicMQ is a complete implementation of the Sun Java Message Service (JMS) v1.0.2, an API for enabling enterprise messaging systems from Java programs.

This book provides the information a Java software developer needs to use the application program interfaces to create SonicMQ client applications.

The sample software provided in source form on the SonicMQ media is the basis for the discussions of features and concepts.

How This Book is Organized

The SonicMQ features are discussed in this programming guide as follows:

- [Chapter 1, “Overview,”](#) discusses the environment and Java constructs that can be used in messaging applications. The basic concepts in this chapter set the groundwork for understanding how to build efficient applications. The service and protection features in SonicMQ are presented in a tabular form with references to other chapters and other books for implementation details.
- [Chapter 2, “Examining the SonicMQ Samples,”](#) takes an in-depth tour through the console-based code samples introduced in the *Getting Started with SonicMQ* manual, focusing on the programming functions and features used.
- [Chapter 3, “SonicMQ Client Sessions,”](#) explores the connection factories, connections, and sessions. The concepts and implementation of the transacted session and transactions are also presented. The parameters and scripts used by various Java clients are detailed.
- [Chapter 4, “Messages,”](#) examines the detailed composition of a message to learn what is required to construct a message, how the data populates the message, and how to manipulate messages.
- [Chapter 5, “Message Producers and Consumers,”](#) presents the scope of the the session objects that produce messages and the session objects that listen, receive, and consume messages.
- [Chapter 6, “Point-to-Point Messaging,”](#) presents the use of server-managed queues and discusses how Point-to-Point contrasts—and how it is similar—to the Publish and Subscribe domain.
- [Chapter 7, “Dynamic Routing Architecture,”](#) describes how global queues provide a richer messaging infrastructure as well as new reasons messages can become undelivered.
- [Chapter 8, “Publish and Subscribe Messaging,”](#) presents the characteristics unique to the broadcast type of messaging, Publish and Subscribe. Durable subscriptions, request-reply mechanisms, message selector semantics, and message listeners are presented in depth.

- In [Chapter 9, “Hierarchical Name Spaces,”](#) presents SonicMQ’s topic hierarchies and how they can be used to streamline access to data.
- [Chapter 10, “Management API,”](#) describes how to run the Broker Manager samples that demonstrate many features of the exposed SonicMQ management API.
- [Chapter 11, “Accessing SonicMQ Through ActiveX/COM Clients,”](#) presents the SonicMQ Java bridge to ActiveX interface with tips, techniques, the sample application, and detailed ActiveX syntax mapping of the `javax.jms` API, exposed `progress.message` API, and some specialized ActiveX commands.
- [Chapter 12, “Lookup of Administered Objects,”](#) shows how programmers can manage and use administered objects with either JNDI or serialized Java objects in a simple file store.

Conventions in This Manual

In this section, you will find a description of the text formatting conventions used in this manual, and a description of notes, warnings, and important messages.

Typographical Conventions and Syntax Notation

This manual uses the following typographical conventions:

- **Bold typeface in this font** indicates keyboard key names (such as **Tab** or **Enter**) and the names of windows, menu commands, buttons, and other SonicMQ user interface elements. For example, “From the **File** menu, choose **Open**.”
Bold typeface is also used to highlight new terms when they are introduced in conceptual and overview sections.
- Monospace typeface is used to indicate text that might appear on a computer screen other than the names of SonicMQ user interface elements, including all of the following:
 - Code examples
 - System output (such as responses, error messages, and so on)

- Filenames and pathnames
- Software component names, such as class and method names

Essentially, `monospace typeface` indicates anything that the computer is “saying,” or that must be entered into the computer in a language that the computer “understands.”

Bold monospace typeface is used to supply emphasis to text that would otherwise appear in `monospace typeface`.

Monospace typeface in italics or ***Bold monospace typeface in italics*** (depending on context) indicates variables or placeholders for values you supply or that might vary from one case to another.

➤ **This symbol and font introduces a multi-step procedure:**

1. This is a first step.
 - 1.1 This is a step within a step.

➤ **This symbol and font introduces a single-step procedure:**

This manual uses the following syntax notation conventions:

- Where command-line examples are provided, a backslash character (\) indicates line continuation. It should not be entered on the actual command line.
- Brackets ([]) in syntax statements indicate parameters that are optional.
- Braces ({ }) indicate that one (and only one) of the enclosed items is required. A vertical bar (|) separates required items.
- Ellipses (. . .) indicate that you can choose one or more of the preceding items.

Note, Important, and Warning Flags

This manual highlights special kinds of information by using shading, placing horizontal rules above and below the text, and using a flag in the left margin to indicate the kind of information.

Note A **Note** flag indicates information that complements the main text flow. Such information is especially needed to understand the concept or procedure being discussed.

Important An **Important** flag indicates information that must be acted upon within the given context in order for the procedure or task to be successfully completed.

Warning A **Warning** flag indicates information that can cause loss of data or other damage if ignored.

Available Documentation

[Table 1](#) lists the documentation supplied with SonicMQ. In addition to the documentation listed in this table, SonicMQ comes with sample files. All documentation is included with the SonicMQ media.

Table 1. The SonicMQ Documentation Set

Document	Description
<i>SonicMQ Documentation Portal</i> (SonicMQ_Help.htm)	Describes and links all SonicMQ online documentation components.
<i>Getting Started with SonicMQ</i>	Presents an introduction to the scope and concepts of the SonicMQ software and its packaging. Lists the features and benefits of SonicMQ in terms of its adherence to the Sun JMS specification and the extensions that make SonicMQ a richer, more useful messaging system.
<i>SonicMQ Installation and Administration Guide</i>	Describes configuration of various SonicMQ client types, clusters, and the message server and data stores. The administration chapters fully document server management using both the command-line interface and the graphical user interface administration tools. Administration of the security interface is fully described.
<i>SonicMQ Programming Guide</i>	Presents the SonicMQ sample applications and then shows how the programmer can enhance the samples, focusing on clients, connections, sessions, messages (including XML), transactions, and hierarchical topics.
<i>SonicMQ Deployment Guide</i>	The first part describes general deployment issues, including security. The second part concerns deployment issues for setting up dynamic routing for a B2B infrastructure.
<i>SonicMQ API Reference</i>	Contains information on the SonicMQ API that supplements the other manuals.
<i>SonicMQ Release Notes</i>	Provides late-breaking information and known issues.

Worldwide Technical Support

Progress Software's support staff maintains a wealth of information at <http://www.sonicmq.com> to assist you with resolving any technical problems that you encounter when installing or using SonicMQ Developer Edition.

From the SonicMQ home page, click on **Developers Exchange** to take advantage of the developer resources such as forums, downloads, tips, whitepapers, and code snippets.

For technical support for the SonicMQ Professional Developer Edition or the SonicMQ E-Business Edition, visit our TechSupport Direct Web page at <http://techweb.progress.com>. When contacting Technical Support, please provide the following information:

- The release version number and serial number of SonicMQ that you are using. This information is listed at the top of the Start Broker console window and might appear as follows:

```
SonicMQ E-Business Edition [Serial Number serial_number]  
Release version_number Build Number n Protocol P22
```

- Your first and last name.
- Your company name, if applicable.
- Phone and fax numbers for contacting you.
- Your e-mail address.
- The platform on which you are running SonicMQ, as well as any other environment information you think might be relevant.
- The Java Virtual Machine you are using.

To determine the JVM you are using, open a console window, go to the directory `SONICMQ_JRE` (default `install-dir\Java\bin`), and issue the command `.\jre -d`.

Table 2 provides information about Progress Software Corporation and its international offices.

Table 2. Progress Software International Offices

Locale, Office Name, and Address	Contact Information
North and Latin America: Progress Software Corporation 14 Oak Park Bedford, MA 01730 USA	Pre-sales: Telephone: 800 477 6473 ext. 4900 e-mail: soni cmqpresales@progress.com Technical Support for Professional Developer Edition and E-Business Edition: Telephone: 781 280 4999 Fax: 781 275 4543 e-mail: support@progress.com
Europe, the Middle East, Africa (EMEA): Progress Software Europe B.V. P.O. Box 8644 Schorpioenstraat 67 3067 GG Rotterdam THE NETHERLANDS	Technical Support for Professional Developer Edition and E-Business Edition: Telephone: 31 10 286 5222 Fax: 31 10 286 5225 e-mail: emeasupport@progress.com
Asia/Pacific: Progress Software Pty. Ltd. 1911 Malvern Road Malvern East, VIC Box 3145, AUSTRALIA	Technical Support for Professional Developer Edition and E-Business Edition: Telephone: 613 9885 0199 e-mail: aussupport@melbourne.progress.com

About SonicMQ

SonicMQ is Progress Software Corporation's implementation of Sun's Java Message Service (JMS) specification that expedites development and deployment of an efficient, secure, and scalable messaging system for business-to-business, networked, and internal integrated applications. SonicMQ makes it possible for organizations to efficiently (and reliably) communicate between disparate business systems over the Internet and meet their time-to-market requirements by delivering the following features:

- Internet-resilient business messaging for the Java platform
- High performance messaging infrastructure
- Reliable transmission of messages regardless of network, hardware, or application failure
- Flexibility in configuring the messaging infrastructure:
 - Clients can be moved around the network without requiring any changes to the messaging application
 - Support for XML message types in addition to the JMS types
- Ease-of-use features make SonicMQ an environment that can be easily learned and deployed

Java Message Service

The Java Message Service (JMS) Version 1.0.2 specification describes portable, efficient standards for a powerful, extensible messaging service. The JMS specification pointedly leaves some functionality—such as load balancing, fault tolerance, error notification, administration, security, wire protocol, and message repository—to the provider of the messaging server.

SonicMQ implements this functionality and provides a level of abstraction to developers, who can concentrate on creating business logic.

JMS: Key Component of the Java Platform for the Enterprise

Sun Microsystems announced a plan in early 1997 to deliver nine Java APIs that would enable a vendor-neutral computing infrastructure capable of integrating Java with virtually every significant enterprise computing service.

JMS would provide asynchronous communications to avoid the problems synchronous communications—such as RMI and CORBA—were experiencing in the uncontrollable Internet environment. Javasoft provided a reference implementation in late 1998, noting that implementers of the JMS specification would need to match the security, reliability, fault-tolerance, and manageability of existing mainframe messaging services before enterprise acceptance would be considered.

The JMS specification notes that it does not address load balancing, fault tolerance, error notification, administration, security, and repositories.

JMS 1.0.2 Specification

On November 5, 1999, Sun introduced version 1.0.2 of the JMS specification. Many of the changes to the content of the specification document describe more precisely some aspects of the JMS functionality. These ambiguities were interpreted correctly in the previous releases of SonicMQ because of the communication between the Progress Software and Sun development teams. Other changes in the JMS 1.0.2 are changes in programming syntax and behavior of applications that use specified techniques.

Java Development Environment

SonicMQ is delivered with a Java run-time environment (JRE) consisting of a Java Virtual Machine (JVM) that is sufficient to support the Java-based installer and the demonstration of SonicMQ samples running against a default database.

You might choose to use a different JVM for use with SonicMQ. In order to deploy SonicMQ applications, you need a JVM appropriate for your target client.

Important See the *SonicMQ Release Notes* in the docs folder of your SonicMQ installation to get detailed information about how to get the JVM that is appropriate for your platform, operating system, database, and toolset.

Programming Concepts

The design of SonicMQ provides full implementation of the Java Message Service (JMS) specification with additional features that comprise a solution that is resilient enough for Internet e-commerce in major enterprises.

Messaging involves the loose coupling of applications. This is accomplished by maintaining an intelligent message server structure. A client can establish one or more connections to a message server.

Clients Connect to the SonicMQ Message Server Architecture

In [Figure 1](#), SonicMQ's hub-and-spoke architecture considers every entity in the messaging service topology to be a client except the message server—the entity to which every client connects and thereby provides connection services to every other client.

The SonicMQ communication layer abstracts developers from the plumbing of the underlying network, freeing them to concentrate on constructing business logic in Java applications.

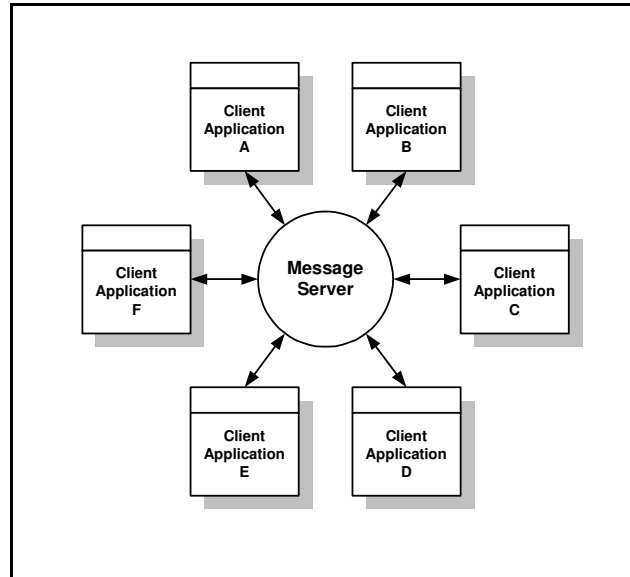


Figure 1. Message Server Is a Hub for SonicMQ Client Applications

The SonicMQ **Message Server** in [Figure 1](#) goes by different names under some circumstances. As the richness of the complete messaging architecture unfolds, you will see that the message server can join with other messaging servers to form **clusters**. The clusters and standalone message servers are nearly equivalent when looked at as **routing nodes**. Within SonicMQ, the message server is often called a **broker**.

SonicMQ Is a JMS Provider

The components that are needed to implement and manage a JMS application are supplied by the **JMS provider**. This includes, as shown in [Figure 2](#), the JMS Client API and the SonicMQ Client Run Time accessed from within the client application, the communications layer between the client and the message server architecture—repositories (message, security, and

configuration), and administrative tools for managing clusters, security, administered objects, and the message servers.

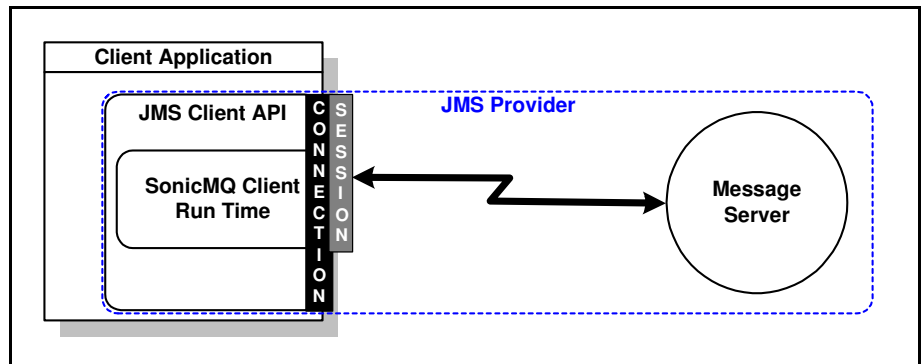


Figure 2. Client Application Using the SonicMQ JMS Provider

SonicMQ Messaging Models

There are two **messaging models** (sometimes referred to as **domains**) in SonicMQ. When a connection is created between a client and a message server, the requested messaging model is declared. The connection is dedicated to the selected messaging model:

- Point-to-Point (PTP)** — A JMS domain where the producer of a message sends a message to a specified static queue at a message server. While many prospective recipients could be listening to or even browsing the queue, when a receiver elects to accept a queued message, the message is considered delivered. No other recipient will thereafter be able to access that message. PTP is a *one-to-one* form of communication.
- Publish and Subscribe (Pub/Sub)** — A JMS domain where the producer of a message sends the message to a specified topic at the message server. Pub/Sub is referred to as *one-to-many* or *broadcast* because there could be zero to many subscribers for a given topic who will each receive the one message that was sent.

SonicMQ Objects and Their Relationships

The view presented in [Figure 3](#) is derived from the SonicMQ Explorer, a graphical client that handles administrative tasks, one of which is examining objects, attributes, and events in SonicMQ. The SonicMQ Explorer will be called on to help you visualize the programming mechanisms described in this guide. See the *SonicMQ Installation and Administration Guide* for more about the SonicMQ Explorer.

[Figure 3](#) presents the primary messaging objects in SonicMQ and their context:

- Clients create **Connections** to **Message Servers** under one of the two JMS domains, Point-to-Point (PTP) or Publish and Subscribe (Pub/Sub).
- Clients create **Sessions** within an established **Connection**.
- Clients create **Destinations**.
For a Point-to-Point domain, the **Destinations** are **Queues**.
- Clients create **Message Producers**. For a Point-to-Point domain:
 - **Message Producers** are **Senders** to **Queues**.
 - **Senders** produce **Messages** to **Queues**.
- Clients create **Message Consumers**. For a Point-to-Point domain:
 - **Message Consumers** are **Receivers** from **Queues**.
 - **Receivers** consume **Messages** from **Queues**.

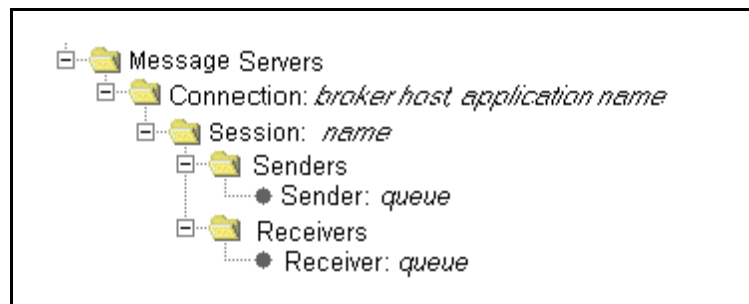


Figure 3. SonicMQ Object Relationships

SonicMQ Object Model

Figure 4 describes the SonicMQ object model in terms of the objects in the Point-to-Point paradigm.

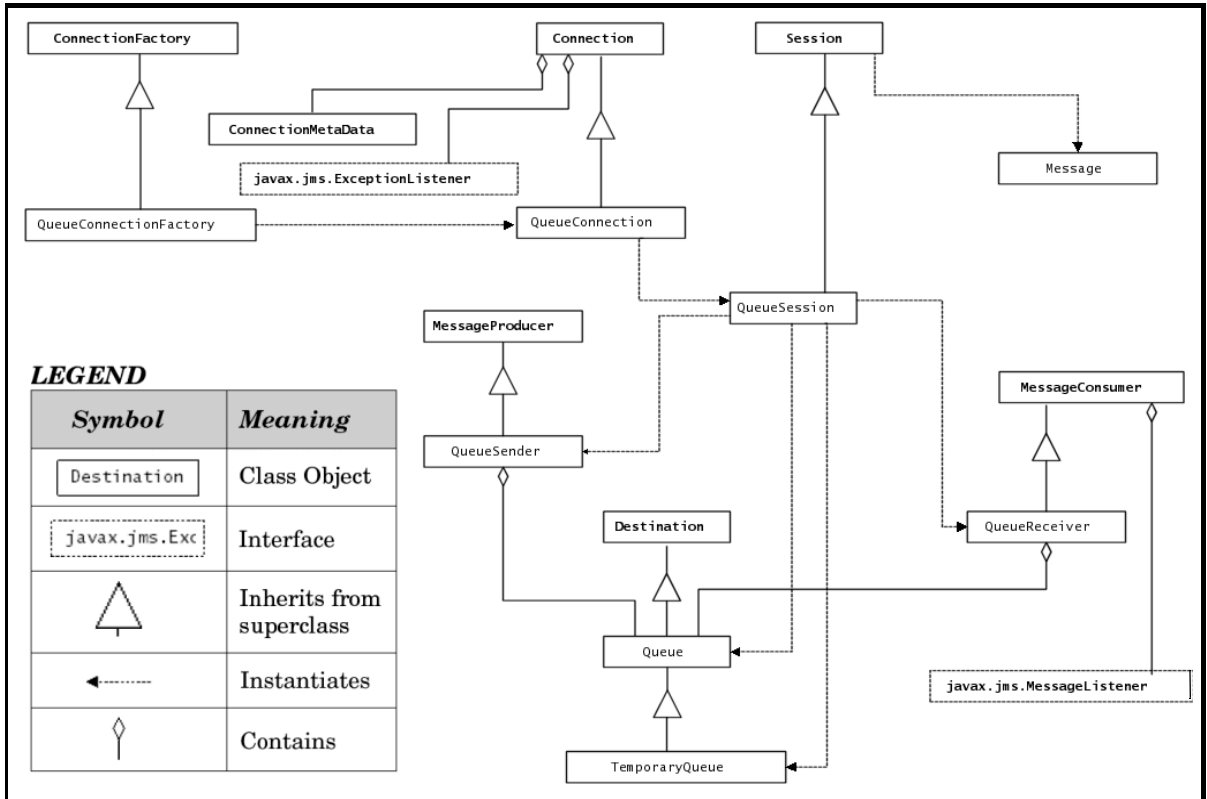


Figure 4. JMS Object Model for the Point-to-Point Domain

Some examples of object relationships are:

- The QueueSession is created by the QueueConnection.
- The QueueSession inherits from the Session.
- The QueueSession creates the QueueSender, the Queue, the QueueReceiver, and the TemporaryQueue.
- The MessageConsumer contains the MessageListener.

Connections and Sessions

An active connection to SonicMQ is a conduit for communication. Each connection is a single point for all communications between the client application and the message server.

While each connection between a client and a message server is a single, synchronous communication, the application can use multiple sessions and asynchronous listeners to minimize the risk of blocking situations where an application is dedicated to waiting.

A connection is dedicated to only one of the messaging paradigms:

- **Point-to-Point (PTP)** — Messaging is *one-to-one* because only one receiver will get the message. The principal PTP interfaces are shown in [Figure 5](#).

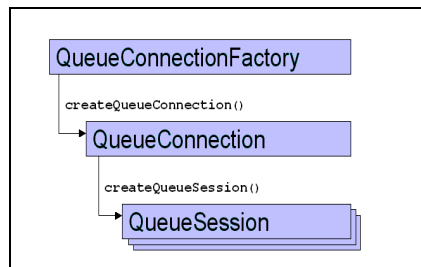


Figure 5. Principal Interfaces for Point-to-Point

- **Publish and Subscribe (Pub/Sub)** — Messaging is *one-to-many* or *broadcast* because there could be *zero-to-many* subscribers for a given topic who will each receive the one message that was sent. The principal Pub/Sub interfaces are shown in [Figure 6](#).

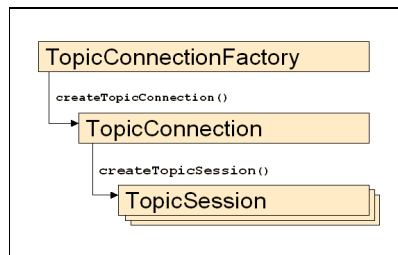


Figure 6. Principal Interfaces for Publish and Subscribe

Producers and Consumers

Entities that create messages and then output the message are **producers**. Entities that actively look for messages that are available are **consumers**.

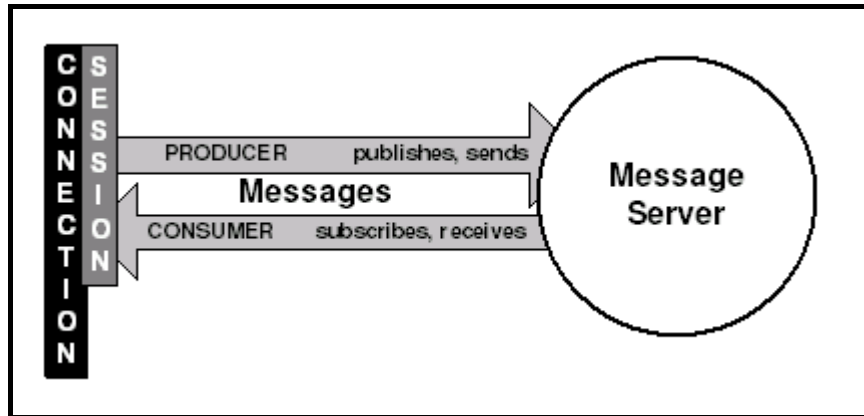


Figure 7. Message Producers and Message Consumers

The client application is the **producer** when:

- **sending** a message to a queue (PTP), or
- **publishing** a message to a topic (Pub/Sub)

The client application is the **consumer** when:

- **receiving** messages from a queue (PTP), or
- **subscribing** to a topic (Pub/Sub) where messages are published

A session can be both a producer and a consumer

To learn about the message server architecture and functionality, see the *SonicMQ Installation and Administration Guide*.

Quality of Service and Protection

Some messages are simple and transitory, and they are broadcast to prospective recipients who might or might not be paying attention. These message might contain information that is timely and important but not particularly confidential. An example is stock quotes. The data is public information that is considered valuable when it is disseminated promptly and verifiable when significant risk might be associated with the information it carries. Here, performance takes precedence.

Messages that represent the other extreme where the anticipated services and protection are paramount include bank wire transfers, where encryption, security, and logging processes are an integral part of mutually assured confidence in the message. Communication that is certifiable, auditable, consistent, and fully credentialed provides the quality of service and the quality of protection that is expected. Performance is important, but not as an alternative to quality.

All the SonicMQ message services and protection are available to both the PTP and Pub/Sub messaging models.

The services and protection that are described in this guide—together with some of the services controlled by the message server's administrator—are described in [Table 3](#).

Table 3. Services and Protection Available in SonicMQ Messaging

Service	Technique	Process	Reference
ENCRYPTED Content is encrypted.	Independent encryption mechanisms.	Body is appended after it has been encrypted, providing assurance that a message is protected even if the connection is insecure.	Private encryption methods can be applied before the message is presented to the messaging-enabled application.
SECURE TRANSPORT Protocol is secure.	Connection protocol parameter.	Parameter is set when creating connection.	See the “ConnectionFactory” on page 93 for information about choosing protocols. See the <i>SonicMQ Installation and Administration Guide</i> for information about ConnectionFactories and protocols.
AUTHENTIC PRODUCER Producer is accepted by the security database.	Security enforced through authentication of user name and password at time of connection.	If the Administrator implemented the security database, the administrator sets up users, passwords, and permissions.	See the <i>SonicMQ Installation and Administration Guide</i> for information about authentication and authorization of producers (PTP senders and Pub/Sub publishers) and Access Control Lists (ACLs).
AUTHORIZED PRODUCER Producer has permission to produce is authorized to produce to specified destination.	Security enforced through Access Control Lists (ACLs).	Administrator sets user authorization to produce to specific hierarchies of destinations.	

Table 3. Services and Protection Available in SonicMQ Messaging (*continued*)

Service	Technique	Process	Reference
<p>ACKNOWLEDGED PRODUCER</p> <p>Server acknowledges receipt of messages from producer.</p>	<p>Synchronous block released after receipt at server.</p>	<p>Automatic when sending a message.</p>	
<p>INTEGRITY</p> <p>Server assures that the message content delivered to the consumer is the identical to what the producer sent.</p>	<p>Administrator function. Message body is digested and the digest accompanies the message to enable integrity checking.</p>	<p>Administrator setting on destination is i n t e g r i t y.</p>	<p>See the <i>SonicMQ Installation and Administration Guide</i> for information about administrator settings for privacy and integrity.</p>
<p>PRIVACY</p> <p>Server assures that the message is read only by the intended consumers and that the message has integrity.</p>	<p>Administrator function. Encrypts the message (privacy) before creating a digest for proof of integrity.</p>	<p>Administrator setting on destination is p r i v a c y (includes the services of i n t e g r i t y).</p>	
<p>PERSISTENT</p> <p>Message persists in server storage.</p>	<p>Delivery mode uses the P E R S I S T E N T option.</p>	<p>Set option in publish or send command. The message server never allows messages to be lost in the event of a network or system failure. Non-persistent messages are volatile in the event of a server failure.</p>	

Table 3. Services and Protection Available in SonicMQ Messaging (*continued*)

Service	Technique	Process	Reference
<p>REDELIVERY</p> <p>Consumer might receive unacknowledged message again.</p>	<p>Message server sets JMSRedelivered field to true when service is interrupted while waiting for a consumer acknowledgement.</p>	<p>Must be checked and acted on by the consumer.</p>	<p>See “Recover” on page 102. See also “connect.start” on page 107.</p>
<p>DURABLE INTEREST</p> <p>Pub/Sub consumers, Subscribers, can establish a durable interest in a topic with a message server.</p>	<p>An application uses the session method createDurableSubscriber with the parameters <i>topic</i>, <i>subscriptionName</i>, <i>messageSelector</i>, and a <i>noLocal</i> option.</p>	<p>Server retains messages for durable subscriber, using the <i>userName</i>, and <i>clientId</i> of the connection plus the <i>subscriptionName</i> to index the subscription.</p> <p>Note that NON_PERSISTENT messages are still at risk in the event of server failure. Note also that messages expire normally even if durable subscriptions are unfulfilled.</p>	<p>See “Reliable, Persistent, and Durable Messaging Samples” on page 60. See also “Durable Subscriber” on page 196.</p>
<p>PRIORITY</p> <p>Messages sent with higher priority can be expedited.</p>	<p>Producer sets the message header value JMSPriority to an <i>int</i> value 0 through 9 where 4 is the default</p>	<p>Server checks message priority and handles appropriately. Priority values of 5 through 9 are expedited.</p>	<p>See also “Message Management by the Message Server” on page 142.</p>

Table 3. Services and Protection Available in SonicMQ Messaging (*continued*)

Service	Technique	Process	Reference
<p>EXPIRATION</p> <p>Messages are available until the expiration time.</p> <p>Based on GMT.</p>	<p>Producer sets <code>time-to-live</code> value, then includes the value at moment of publish/send.</p>	<p>Server receives message with <code>JMSExpiration</code> date-time set to the <code>JMSTimestamp</code> date-time plus the <code>time-to-live</code> value.</p>	<p>See “Create the Message Type and Set Its Body” on page 139.</p> <p>See also “Message Management by the Message Server” on page 142.</p>
<p>REQUEST MECHANISM</p> <p>Producer can request a reply from the consumer.</p>	<p>Message header field <code>JMSReplyTo</code> has a string value that indicates the topic where a reply is expected. The <code>JMSCorrelationID</code> can indicate a reference string whose uniqueness is managed by the producer.</p>	<p>Carried through to consumer, but the consumer application must be coded to look at the <code>JMSReplyTo</code> field and then act.</p> <p>Producer could be synchronously blocked waiting for reply message at temporary topic.</p> <p><code>TopicRequestor</code> object creates a temporary topic for the reply.</p>	<p>See “Request and Reply Samples” on page 72.</p> <p>See also “Session Objects” on page 103 and “Reply-to Mechanisms” on page 153.</p>
<p>AUTHENTIC CONSUMER</p> <p>Consumer is accepted by the security database.</p>	<p>Security enforced through authentication of username and password at time of connection.</p>	<p>Administrator initialized and started security database at server; administrator sets up user and password.</p>	<p>See the <i>SonicMQ Installation and Administration Guide</i> for more about authentication and authorization of consumers (PTP receivers and Pub/Sub subscribers) and Access Control Lists (ACLs).</p>
<p>AUTHORIZED CONSUMER</p> <p>Consumer is authorized to consume from a specified destination.</p>	<p>Security enforced through ACLs.</p>	<p>Administrator sets user authorization to specific hierarchies of destinations.</p>	<p>(ACLs).</p>

Table 3. Services and Protection Available in SonicMQ Messaging (*continued*)

Service	Technique	Process	Reference
<p>ACKNOWLEDGED CONSUMPTION</p> <p>Consumer acknowledges receipt to server.</p>	Acknowledgement type for the session was set when the session was created.	Functions automatically to perform the specified type of acknowledgement for all messages consumed in that session.	See “Acknowledgement Mode” on page 101.
<p>Client acknowledges receipt of received messages when session parameter is <code>CLIENT_ACKNOWLEDGE</code> then when client calls <code>acknowledge()</code></p>	Explicit call by consumer.	Manual.	
<p>REPLY MECHANISM</p> <p>Consumer replies to the producer’s request for reply.</p>	Consumer reacts to a <code>JMSReplyTo</code> request by producing a message to the topic name in the <code>JMSReplyTo</code> field.	Programmatic procedure where the consumer publishes a reply. The content of the reply is not specified. Typically the <code>JMSCorrelationID</code> would be replicated.	See “Request and Reply Samples” on page 72. See also “Session Objects” on page 103 and “Reply-to Mechanisms” on page 153.
<p>DEAD MESSAGE QUEUE</p> <p>Sender/publisher can set properties to either or both re-enqueue undelivered messages and send an administrative notice.</p>	Set the properties that tell the message server to provide special handling when the message is declared dead.	Programmatic procedure where the sender chooses to set the property <code>JMS_SonicMQ_preserveUndelivered</code> to <code>true</code> to store the dead message until handled and to set the property <code>JMS_SonicMQ_notifyUndelivered</code> to <code>true</code> to send a notification to the message server’s administrator.	See “Message Properties” on page 127. See also Chapter 7, “Dynamic Routing Architecture.”

SonicMQ Clients

There are several types of SonicMQ clients. The client Java archives are copied in libraries to enable bridges, proxy servers, servlet engines, and JavaBeans. This book presents techniques and interfaces to enable class files to run in a console session as well as ActiveX/COM clients and Java Applet clients.

ActiveX/COM Client

SonicMQ can work as an ActiveX/COM control, providing developers with an interface that makes the SonicMQ JMS API available in popular Windows development environments.

Using JMS functionality delivers the advantages of messaging to both new and established applications through familiar developer environments such as Visual Studio and run-time environments such as Microsoft Office, Internet Explorer, and Lotus® Notes—to name a few.

See [Chapter 11, “Accessing SonicMQ Through ActiveX/COM Clients,”](#) for more information.

Java Applet Client

SonicMQ can work in a Java applet running in a browser context to invoke classes that implement JMS functionality.

About SonicMQ Samples

The samples provided with the SonicMQ product, first explored in the *Getting Started with SonicMQ* manual, are now viewed in terms of their functionality. These samples demonstrate programmatic interaction between applications.

When you run the samples, consider that the standard input and standard output displayed in the console could be data flows to and from a whole range of applications and Internet-enabled devices such as:

- **Application software** for accounting, auditing, reservations, online ordering, credit verification, medical records, and supply chains
- **Information appliances** such as beepers, cell phones, fax machines, and Personal Digital Assistants (PDAs)
- **Real-time devices** with embedded controls such as monitor cameras, medical delivery systems, climate control systems, and machinery
- **Distributed knowledge bases** such as collaborative designs, service histories, medical histories, and workflow monitors

Note The samples in this chapter assume that you are not using a security database, which is the default SonicMQ setup. Exercises are provided at the end of the chapter that detail how to reconfigure the database for security and how to enter the user names and password that security will demand. Without security, user names in the samples are arbitrary strings. Still, the names cannot contain the reserved characters, period (.), pound (#), dollar (\$), or asterisk (*).

SonicMQ Samples

The SonicMQ samples present basic features of SonicMQ, categorized as follows:

- **Chat and Talk Samples** — The basic messaging functions are presented by producing and consuming messages in both domains:
 - `Talk` (PTP), `Chat` (Pub/Sub)
- **Transaction Samples** — Transactions are shown in both domains in application windows that reveal how the producers and consumers of the transacted messages see the messages flow:
 - `TransactedTalk` (PTP), `TransactedChat` (Pub/Sub)
- **Additional Message Types** — To simplify input, the preceding examples are Text messages. The following samples display two other common message types in the messaging domains:
 - **XMLMessages** — `XMLTalk` (PTP), `XMLChat` (Pub/Sub)
 - **MapMessages** — `MapTalk` (PTP)
- **Message Traffic Monitors** — These samples provide views of message traffic in ways that are characteristic of their messaging domain:
 - **Messages on the Queue** — `QueueMonitor` (PTP)
 - **Messages to Subscribers** — `MessageMonitor` (Pub/Sub)
- **Reliable, Persistent, and Durable Messaging** — These samples demonstrate techniques that can enhance the Quality of Service. Reliable connections show how to keep connections active in both domains. Persistent storage shows how the message server's PTP safety net, the Dead Message Queue, can trap undelivered messages. Durable subscription shows how a Pub/Sub subscriber can have messages held for them. The samples in this category are:
 - **Reliable Connection** — `ReliableTalk` (PTP), `ReliableChat` (Pub/Sub)
 - **Persistent Storage** — `DeadMessages` (PTP)
 - **Durable Subscription** — `DurableChat` (Pub/Sub)

- **Request and Reply** — These transacted examples show the mechanisms for the producer requesting a reply and the consumer fulfilling that request:
 - **Originator’s Request** — `Requestor` (PTP, Pub/Sub)
 - **Receiver’s Response** — `Replier` (PTP, Pub/Sub)
- **Selection and Wildcards** — The message selector samples use SQL syntax to qualify the messages that are visible to an application while the `HierarchicalChat` sample uses template characters to subscribe to a set of topics that is qualified when messages are published:
 - **Message Selection** — `SelectorTalk` (PTP), `SelectorChat` (Pub/Sub)
 - **Wildcards** — `HierarchicalChat` (Pub/Sub)
- **Test Loop** — This sample makes it easy to get a look at how quickly messages can be sent and received in a test loop:
 - **Queue Test Loop** — `QueueRoundTrip` (PTP)

Other Samples Available

There are several other SonicMQ samples that require special setup to explore them. These samples are described in other SonicMQ documents:

- **ActiveX/COM** — The ActiveX/COM sample, `Chat.frm`, requires the Windows Visual Basic development and run-time environments plus a few setup steps. See [Chapter 11, “Accessing SonicMQ Through ActiveX/COM Clients,”](#) for more information.
- **Dynamic Routing Queues** — When routing queues are established across message servers, messages are dynamic. The `GlobalTalk` (PTP) sample demonstrates dynamic routing queues once you have an appropriate setup. See the *SonicMQ Deployment Guide* for information about this sample.
- **Management API** — The exposed administrative methods make it possible to create applications that perform management functions. There are several samples of management applications packaged with SonicMQ. To see how to run these samples, see [Chapter 10, “Management API.”](#)

Extending the Samples

After reviewing the sample applications, you are encouraged to explore some variations:

- **Change the source files** — You can edit the source files, compile the changed file, and then run the applications again to observe the effect. Some ideas are presented as exercises:
 - Using a common topic for two samples.
 - Observing how different messaging behaviors affect round-trip times.
 - Modifying the `MapMessage` to use other data types.
 - Modifying the `XMLMessage` to show more data.
- **Initializing the message server database for security** — The impact of security is apparent throughout the samples when user access and destination access are controlled by administrated security.

How Security Impacts Client Activities

Security provides the high quality of protection and access by applications that is expected in enterprise applications. The section [“Quality of Service and Protection” on page 32](#) presents an overview of the features and functions of security. But unless the message server database initializes to manage security, security is not enabled.

The samples in this chapter do not initialize the security database so that you can begin exploring the messaging features without having to first set up security objects for:

- **User authentication** — When security is activated, only defined usernames are allowed to connect to the message server.
- **User authorizations** — The administrator can control a user’s ability to perform actions such as subscribing to a topic and reading from queues.

[“Extending the Samples” on page 79](#) explores what you need to do to implement a SonicMQ sample under a secure environment.

Running the SonicMQ Samples

Starting the Message Server Under Windows, Linux, or UNIX

Be sure the SonicMQ message server is running before executing any of the SonicMQ client samples.

Note If this is the first time you are running SonicMQ, you should not have to set up and initialize the database or adjust the message server's `broker.ini` file. See the *SonicMQ Installation and Administration Guide* for `broker.ini` settings.

➤ **To start the message server process from the Windows Start menu:**

□ Choose **Start > Programs > Progress SonicMQ > Start Broker**.

➤ **To start the message server process from a Linux or UNIX console window:**

□ In a new console window set to the SonicMQ install directory, type `startbr.sh` and press **Enter**.

The message server starts. The console window is dedicated to the process and, when running, displays:

```
SonicMQ Broker started, now accepting tcp connections on port 2506...
```

Important You can minimize the console window. Closing the window, however, stops the message server.

The samples default to `localhost:2506`—a message server using port 2506 on the same system, `localhost`. If you use a different host or port, you need to specify the `host:port` parameter when you start each sample; for example:

```
..\..\SonicMQ Chat -u Market_Maker -b Eagle:2345
```

Client Console Windows

Each application instance is intended to run in its own console window with the current path in the selected sample directory. There are conventions that you must follow depending on the platform:

- **Windows** — The scripts defer to Windows conventions.
- **Linux and UNIX platforms** — Instead of using `.bat` files, use the `.sh` file at the same location. Substitute forward slash (`/`) wherever back slash (`\`) is used as a path delimiter. Any sourcing is handled in the shell scripts.

Note Consider all text to be case-sensitive. While there may be some platforms and names where case is not distinguished, it is good practice to always use case consistently.

Using the Sample Scripts

A universal script handler is installed at the `Samples` directory level. This script, `SonicMQ.bat` (`.sh` under Linux and UNIX), does the following:

- Points to the Java executable used by SonicMQ
- Sets the `CLASSPATH` for the Java runtime and SonicMQ `.jar` files.
- Invokes the executable, its parameters, and a list of variables

The script is suitable for the basic samples provided, but you might have to adjust it if you use long parameter lists. Standard invocation of the script from a sample folder is two levels down.

Important When you modify the original sample files, you can use the techniques described above to set up a universal compiler script. Replicate and modify `SonicMQ.bat` (`.sh` under Linux and UNIX) to something like `SonicMQ_javac.bat` (`.sh` under Linux and UNIX) and then confirm that `javac.exe` (or the path to your preferred compiler) is in the script.

Using the SonicMQ Explorer

You can use the SonicMQ Explorer to see the parameters and action events available in SonicMQ.

► **To start the SonicMQ Explorer under Windows:**

- Choose **Start > Programs > Progress SonicMQ > Explorer**.

The SonicMQ Explorer window opens.

► **To start the SonicMQ Explorer under UNIX:**

- In a console window positioned in the SonicMQ working directory, type `explorer.sh` and press **Enter**.

The SonicMQ Explorer window opens.

► **To review or set up the default queues in the SonicMQ Explorer:**

2. Click on **Message Brokers** in the Explorer tree.
3. Enter the Broker Host you are using, typically `localhost:2506`.
4. Enter any Connect ID text such as `Conn1` then choose **Connect**.
5. Click on the message server you just connected to: `localhost:2506:Conn1`.
6. Click on **Queues**, then verify that the queues are those in [Figure 8](#).

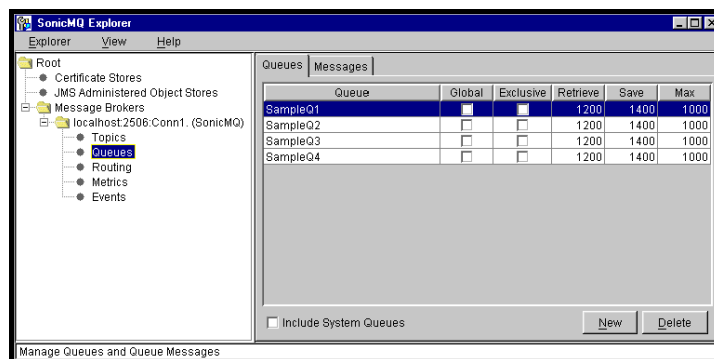


Figure 8. Using the Explorer to Maintain the Default Queues

See the *SonicMQ Installation and Administration Guide* for details about the SonicMQ Explorer and maintaining queues.

Chat and Talk Samples

The fundamental differences between Pub/Sub and PTP are presented in the Chat and Talk samples.

Chat Application (Pub/Sub)

In the Chat application, whenever anyone sends a text message to a given topic, all active applications running Chat receive that message as subscribers to that topic. This is the most basic form of publish and subscribe activity.

► **To start Chat sessions:**

1. Open a console window to the TopicPubSub\Chat folder, then enter:
`.. \.. \SonicMQ Chat -u OTC_Ticker`
2. Open another console window to the TopicPubSub\Chat folder, then enter:
`.. \.. \SonicMQ Chat -u Market_Maker`

► **To Chat:**

1. In one of the Chat windows, type any text and then press **Enter**. The text is displayed in both Chat windows, preceded by the Chat name that initiated that text.
2. In the other Chat window, type text and then press **Enter**. The text is displayed in both Chat windows preceded by that Chat name.

The Chat sample shows inter-application asynchronous communications. If subscribers miss some of the messages, they just pick up the latest whenever they re-connect to the message server. Nothing is retained and nothing is guaranteed to be delivered, so throughput is fast.

Talk Application (PTP)

In the Talk application, whenever a text message is sent to a given queue, all active Talk applications are waiting to receive messages on that queue, taking turns as the sole receiver of the message at the front of the queue.

► To start Talk sessions:

The first Talk session receives on the first queue and sends to the second queue while the other Talk session does the opposite.

1. Open a console window to the QueuePTP\Talk folder, then enter:

```
.. \. . \SonicMQ Talk -u Accounting -qr SampleQ1 -qs SampleQ2
```
2. Open another console window to the QueuePTP\Talk folder, then enter:

```
.. \. . \SonicMQ Talk -u Orders -qr SampleQ2 -qs SampleQ1
```

► To Talk:

1. In the Orders window, type any text and then press **Enter**.
The text is displayed in only the Accounting window.
2. In the Accounting window, type text and then press **Enter**.
The text is displayed in only the Orders window.

Reviewing the Chat and Talk Samples

You can extend your exploration of the samples by opening several windows:

- **Chat** — If you run several Chat windows, every window will display the message, including the publisher. You can modify the source code to suppress delivery of a Chat message to its publisher. That Pub/Sub broadcast characteristic could be stopped with a `noLocal` parameter on the `createSubscriber` method. Every subscriber would get everyone else's messages except their own.
- **Talk** — If you run several Talk windows, you will still see only one receiver for any message. Under Talk (PTP), there is only one receiver. Start two more Accounting windows (Accounting1 and Accounting2) then use the Orders window to send 1 through 9, each as a message (1 Enter, 2 Enter. . .). Notice how the receivers take turns receiving the messages.

Samples of Additional Message Types

Most of the SonicMQ samples use the `TextMessage` type because they accept user input in the console windows. Additional message type samples offer some variation from the `TextMessage` to kindle your interest in other message types while still using text input.

XML Messages

XML data definitions with tagged text are rapidly gaining favor as a technique for communicating structured sets of defined data records or transacted message sets over the Internet. The XML parser included with SonicMQ, the IBM® XML Parser for Java Edition, interprets the data using Document Object Model Element nodes. The message receiver window echoes its translation of the XML-tagged code derived from your text entry. For example, if you (as the sender **Catalog_Update**) enter **Item One**, the XML-tagged code is packaged as follows in the sample source file `XMLChat.java`:

```
{
    progress.message.jclient.XMLMessage xMsg =
        ((progress.message.jclient.Session) pubSession).createXMLMessage();
    StringBuffer msg = new StringBuffer();
    msg.append("<?xml version='1.0'?>\n");
    msg.append("<message>\n");
    msg.append("    <sender>" + username + "</sender>\n");
    msg.append("    <content>" + content + "</content>\n");
    msg.append("</message>\n");
    xMsg.setText( msg.toString() );
    publisher.publish( xMsg );
}
```

The tagged message text is well-formed XML:

```
<?xml version="1.0"?>
  <message>
    <sender>"Catalog_Update"</sender>
    <content>"Item One"</content>
  </message>
```

When the message is received, the embedded XML parser is invoked. The message is interpreted to display the DOM nodes:

```
ELEMENT: message
|--NEWLINE
+--ELEMENT: sender
  |--TEXT_NODE: Catalog_Update
|--NEWLINE
+--ELEMENT: content
  |--TEXT_NODE: Item One
|--NEWLINE
```


XML Messages (PTP)

► To start PTP XMLTalk sessions:

The first XMLTalk session receives on the first queue and sends to the second queue while the other session does the opposite.

1. Open a console window to the QueuePTP\XMLTalk folder, then enter:

```
.. \. . \SonicMQ XMLTalk -u QCatalog_Update -qr SampleQ1 -qs SampleQ2
```
2. Open another console window to the QueuePTP\XMLTalk folder, then enter:

```
.. \. . \SonicMQ XMLTalk -u QLocal_Supplier -qr SampleQ2 -qs SampleQ1
```

► To send and receive PTP XMLMessages:

- In the QCatalog_Update window, type text and then press **Enter**.

XML Messages (Pub/Sub)

► To start Pub/Sub XMLChat sessions:

1. Open a console window to the TopicPubSub\XMLChat folder, then enter:

```
.. \. . \SonicMQ XMLChat -u Catalog_Update
```
2. Open another console window to the TopicPubSub\XMLChat folder, then enter:

```
.. \. . \SonicMQ XMLChat -u Local_Supplier
```

► To produce and consume Pub/Sub XMLMessages:

- In the Catalog_Update window, type text and then press **Enter**.

Map Messages (PTP)

A Map message is a message type that transfers a collection of assigned names and their respective values. The names and values are assigned by set methods for the Java primitive data type of the value. The MapMessage name-value pairs are sent in the message body.

For example:

```
mapMessage.setInt("FiscalYearEnd", 10)
mapMessage.setString("Distribution", "Global")
```

```
mapMessage.setBoolean("LineOfCredit", true)
```

You can extract the data from the received message in any order. Use a `get` method to coerce a data value into an acceptable data type.

For example:

```
mapMessage.getShort("FiscalYear-End")
mapMessage.getString("Distribution")
mapMessage.getString("LineOfCredit")
```

► To start MapTalk sessions:

The first MapTalk session receives on the first queue and sends to the second queue, while the other session does the opposite:

1. Open a console window to the `QueuePTP\MapTalk` folder, then enter:
`..\..\SonicMQ\MapTalk -u QAccounting -qr SampleQ1 -qs SampleQ2`
2. Open another console window to the `QueuePTP\MapTalk` folder then enter:
`..\..\SonicMQ\MapTalk -u QAuditing -qr SampleQ2 -qs SampleQ1`

► To send and receive MapMessages:

- In the `QAccounting` window, type text and then press **Enter**.

The message sender packages two items: the username as the `String` sender and the text input into a `String` named content as shown in the source code of the sample `MapTalk.java`:

```
javax.jms.MapMessage msg = sendSession.createMapMessage();
msg.setString("sender", username);
msg.setString("content", s);
```

The message receiver casts the message as a `MapMessage`. If that is unsuccessful, MapTalk reports that an invalid message arrived. The `MapMessage` is decomposed and displayed as shown in the source code of the sample `MapTalk.java`:

```
String sender = mapMessage.getString("sender");
String content = mapMessage.getString("content");
System.out.println(sender + ": " + content);
```

Reviewing the Additional Message Type Samples

In review, these samples show:

- The message type characteristics are identical in PTP and Pub/Sub.
- These messages are limited to capturing a single chunk of text in the console window.
- These messages use the `instanceof` operator to identify and cast the message into an `XMLMessage` or a `MapMessage`.

You could modify the source code of these samples to:

- Create a table of XML data that forms an `XMLMessage`.
- Set some map values to Java primitives in the `MapMessage` and then get the map values, coercing them into acceptable data types.

See the exercises in [“Extending the Samples” on page 79](#) that describe these changes. See also [“Message Type” on page 120](#).

Message Traffic Monitor Samples

These samples each open GUI windows that provides a scrolling array of its contents. The nature of the two monitors underscores fundamental differences between the Publish and Subscribe messaging model and the Point-to-Point messaging model. [Table 4](#) shows these differences.

Table 4. Differences Between QueueMonitor and MessageMonitor

	<i>QueueMonitor</i>	<i>MessageMonitor</i>
What messages are displayed?	Undelivered.	Delivered.
When does the display update?	When you click the Browse Queues button, the list is refreshed.	When a message is published to a subscribed topic, it is added to the displayed list.
When does the message go away?	When the message is delivered (or when it expires.)	When the display is cleared for any reason.
What happens when the message server and monitor are restarted?	Listed messages marked PERSISTENT are stored in the message server database. They are redisplayed when the message server and the QueueMonitor restart and then choose to browse queues.	As messages are listed at the moment they are delivered, there are no messages in the MessageMonitor until new deliveries occur.

QueueMonitor Application (PTP)

The QueueMonitor moves through a queue, listing the active messages it reveals as it traverses the queue.

► **To start QueueMonitor:**

1. Open a console window to the QueuePTP\QueueMonitor folder.
2. Type `..\..\SonicMQ\QueueMonitor` and press **Enter**.

► **To start a Talk session without a receiver:**

1. Open a console window to the QueuePTP\Talk folder.
2. Type `..\..\SonicMQ\Talk -u RFP -qs SampleQ1` and press **Enter**.

► **To enqueue messages and then browse the queue:**

1. In the Talk window, type some text and then press **Enter**. Repeat a few times.
2. In the QueueMonitor Java window, click **Browse Queues** to scan the queues and display their contents. The QueueMonitor appears similar to the window shown in [Figure 9](#).

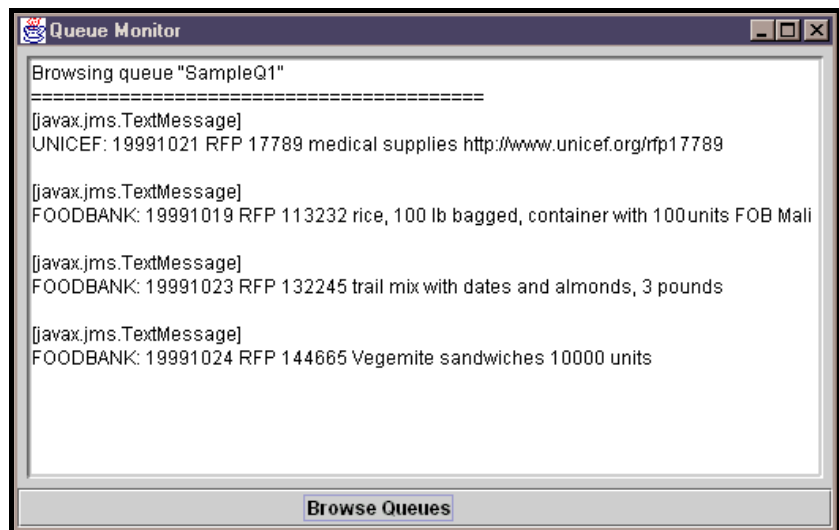


Figure 9. QueueMonitor Window

➤ **To receive the queued messages:**

The messages that are waiting on the queue will get delivered to the next receiver who chooses to receive from that queue.

Warning If you do not perform this procedure the stored messages will be received in the next application that receives on that queue.

1. In a console window, press **Ctrl+C**. The application stops.
2. Type `.. \. \SonicMQ Talk -u FlushQ1 -qr SampleQ1` and press **Enter**.
The enqueued messages are delivered to the queue receiver.

➤ **To stop the sample:**

1. In the console window, press **Ctrl+C**. The application stops.
2. In the QueueMonitor window, click the close button.

MessageMonitor Application (Pub/Sub)

An example of a supervisory application with a graphical interface is MessageMonitor where the application listens for any message activity—by subscribing to all topics in the topic hierarchy—and then displays each message in its window.

➤ **To start MessageMonitor:**

1. Open a console window to the TopicPubSub\MessageMonitor folder, and then enter: `.. \. \SonicMQ MessageMonitor`
The MessageMonitor Java window opens.

➤ **To run a Chat session to send messages to the MessageMonitor**

1. Open a console window to the TopicPubSub\Chat folder, then enter:
`.. \. \SonicMQ Chat -u Chatter`

2. Type any text and then press **Enter**. The text is displayed in the Chat windows, and the **MessageMonitor** window. If you send more messages, each one appends to the list displayed, as shown in [Figure 10](#).

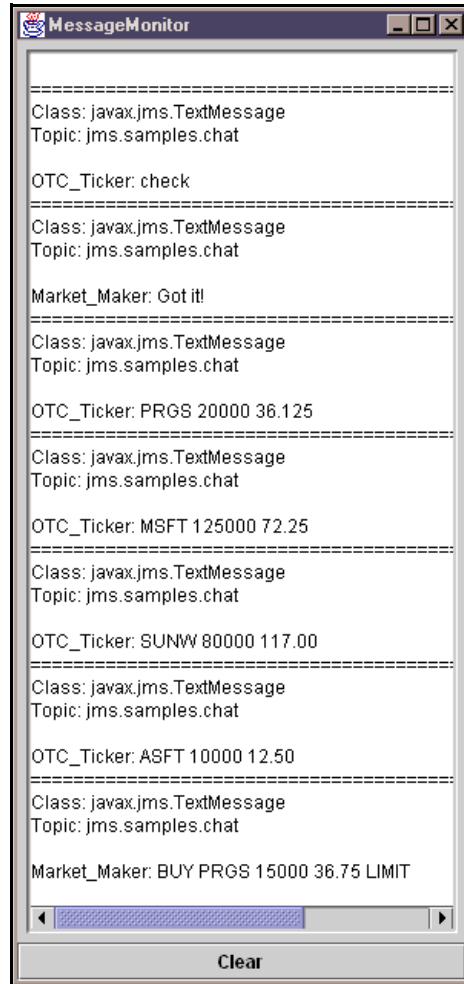


Figure 10. Message Monitor Window

3. Click the Clear button. The list is emptied.

Transaction Samples

Transacted messages are a group of messages that form a single unit of work. Much like an accounting transaction made up of a set of balancing entries, a messaging example might be a set of financial statistics where each entry is a completely formed message and the full set of data comprises the update.

A session is declared as **transacted** when the session is created. While producers—PTP Senders and Pub/Sub Publishers—produce messages as usual, the messages are stored at the message server until the message server is notified to act on the transaction by delivering or deleting the messages. The programmer must determine when the transaction is complete:

- Call the method to **commit** the set of messages. The session `commit` method tells the message server to sequentially release each of the messages that have been cached since the last transaction. In this sample, the commit case is set for the string `OVER`.
- Call the method to **roll back** the set of messages. The session `rollback` method tells the message server to flush all the messages that have been cached since the last transaction ended. In this sample, the rollback case is set for the string `00PS!`.

TransactedTalk Application (PTP)

➤ **To start TransactedTalk sessions:**

The first TransactedTalk session receives on the first queue and sends to the second queue, while the other session does the opposite.

1. Open a console window to the `QueuePTP\TransactedTalk` folder, then enter:
`.. \. \SonicMQ TransactedTalk -u Accounting -qr SampleQ1 -qs SampleQ2`
2. Open another console window to the `QueuePTP\TransactedTalk` folder, then enter:
`.. \. \SonicMQ TransactedTalk -u Operations -qr SampleQ2 -qs SampleQ1`

➤ **To build a PTP transaction and commit it:**

1. In a TransactedTalk window, type any text and then press **Enter**. Notice that the text is *not* displayed in the other TransactedTalk window.
2. Type more text in that window and then press **Enter**. The text is still not displayed in the other TransactedTalk window.

3. Type **OVER** and then press **Enter**. All the messages you sent to a queue are delivered to the receiver. Subsequent entries will form a new transaction.

➤ **To build a PTP transaction and rolling it back:**

1. In one of the `TransactedTalk` windows, type text and then press **Enter**.
2. Type more text in that window and then press **Enter**.
3. Type **OOPS!** and then press **Enter**. Nothing is published.

All messages are removed from the message server. Subsequent messages will form a new transaction. Any messages you re-send will be re-delivered.

TransactedChat Application (Pub/Sub)

➤ **To start Pub/Sub TransactedChat sessions:**

1. Open a console window to the `TopicPubSub\TransactedChat` folder, then enter: `.. \.. \SonicMQ TransactedChat -u Sales`
2. Open another console window to the `TopicPubSub\TransactedChat` folder, then enter: `.. \.. \SonicMQ TransactedChat -u Audit`

➤ **To build a Pub/Sub transaction and commit it:**

1. In the `Sales` window, type any text and then press **Enter**. Notice that the text is *not* displayed in the `Audit` window.
2. Type more text in the `Sales` window and then press **Enter**. The text is still not displayed in the `Audit` window.
3. Type **OVER** and then press **Enter**. All of the messages now display in sequence in the `Audit` window.

All of the lines you had published to a topic are delivered to subscribers. Subsequent entries will form a new transaction.

➤ **To build a Pub/Sub transaction and roll it back:**

1. In the `Sales` window, type text and then press **Enter**.
2. Type more text in that window and then press **Enter**.
3. Type **OOPS!** and then press **Enter**. Nothing is published.

All messages are removed from the message server. Subsequent entries will form a new transaction. Any messages you resend will be redelivered.

Reviewing the Transaction Samples

In review, the transaction samples show:

- The transaction scope is between the client in the JMS session and the message server. When the message server receives commitment, the messages are placed onto queues or topics in the order in which they were buffered but with no transaction controls. Message delivery is normal:
 - **PTP Messages** — The order of messages in the queue is maintained with adjustments for priority differences but there is no guarantee that—when multiple receivers are active on the queue—a `QueueReceiver` will receive one or more of the sender’s transacted messages.
 - **Pub/Sub Messages** — Messages are delivered in the order entered in the transaction yet influenced by the priority setting of these and other messages, the use of additional receiving sessions, and the use of additional or alternate topics. The messages are not delivered as a group.
- Transactions are a set of messages that is complete only when a command is given. As an alternative, message volume could be reduced by packaging sets of messages. For example, an XML message enables the publisher to send a package of messages and the subscriber to interpret the set of packaged entries as a single message. See the `XMLChat` example for details.
- While most of the samples use two sessions—a producer session to listen for keyboard input and send messages, and a consumer session to listen for messages and receive them—the transacted samples set only the producer session as transacted so that committing or rolling back impacts only the sent messages.

Changing the receiver behavior has no real effect on non-durable Pub/Sub messages but causes an interesting behavior in PTP: When you rollback receipt of messages, the message listener sees the messages again and then simply receives them again. Rolling back a transacted consumer session causes the messages to be re-delivered.

You can explore this behavior by modifying `TransactedTalk.java` to set the receive session to be transacted, like this:

```
receiveSession = connect.createQueueSession  
    (false, javax.jms.Session.AUTO_ACKNOWLEDGE);
```

Then follow the send session commit line and send session rollback line with similar statements for the receive session like this:

```
sendSession.rollback();  
receiveSession.rollback();  
...  
sendSession.commit();  
receiveSession.commit();
```

Start the two sessions described in the `TransactedTalk` sample then run `QueueMonitor` sample. Notice that whether you commit or rollback, no messages stay in the queue. Stop the `TransactedTalk` sessions and then refresh the queue monitor. Note that the messages sent since the last commit were all reinstated in the queue.

For more information, see [“Transacted Sessions” on page 102](#).

Reliable, Persistent, and Durable Messaging Samples

The preceding applications made the same, basic delivery promise: If you are connected and receiving during the message's lifespan, you could be a consumer of this message.

One of the features of SonicMQ is the breadth of services that can be applied to messaging to give just the right **quality of service (QoS)** for each type and category of message.

There are programmatic mechanisms for:

- Increasing the chances that the client and message server are actively connected
- Registering a Point-to-Point sender's interest in routing messages that are undeliverable to a dead message queue and sending notification events to the administrator
- Registering a Pub/Sub subscriber's interest in messages published to a topic even when the subscriber is disconnected

The reliable, persistent, and durable messaging samples explore these features of SonicMQ.

Reliable Connections

The ReliableConnection sample ensures the robustness of the JMS connection by monitoring the connection for exceptions and re-establishing the connection if it has been dropped.

Note The Reliable samples use an aggressive technique (CTRL+C) that emulates an unexpected message server interruption.

An intentional shutdown invokes an administrative **Shutdown** function on the message server. This function is a command in the Explorer tool and the Admin tool. It is also part of the management API that you can review and explore in the Shutdown sample presented in [Chapter 10, "Management API."](#)

In a Talk session, if the message server stopped and you sent a message you would see:

```
Exception in thread "main" java.lang.NullPointerException
    at ... QueueSender.internalSend(QueueSender.java:343)
    at ... QueueSender.send(QueueSender.java:194)
    at Talk.talker(Talk.java:124)
    at Talk.main(Talk.java:287)
```

To ensure higher reliability, both reliable samples use a rich connection setup routine for connection retries and `Thread.sleep(CONNECTION_RETRY_PERIOD)`.

In addition, using the `PERSISTENT` deliveryMode option ensures that messages are logged before they are acknowledged and are non-volatile in the event of a message server failure. Consequently, as shown in [Figure 11](#), the application tries repeatedly to reconnect.

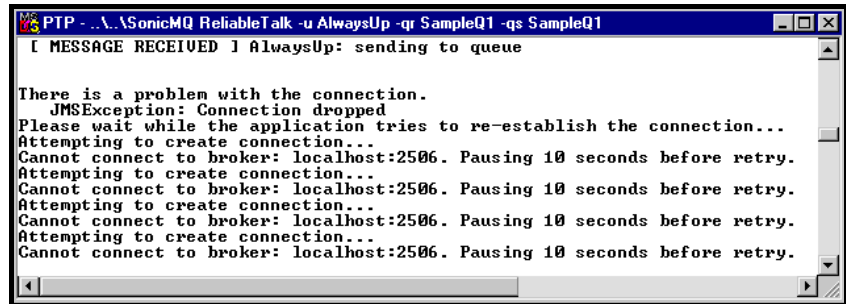
A unique SonicMQ feature monitors the heartbeat of the message server by pinging the message server at a preset interval, letting the thread sleep for a while but initiating reconnection if the message server does not respond. For more information, see [“Using Active Pings to Monitor the Health of the Connection”](#) on page 114.

ReliableTalk Application (PTP)

► **To run the ReliableTalk sample:**

1. Open a console window to the `QueuePTP\ReliableTalk` folder, then enter:
`..\..\SonicMQ\ReliableTalk -u AlwaysUp -qr SampleQ1 -qs SampleQ1`
2. Type text and then press **Enter**. The text is displayed, preceded by the user name that initiated that text. The message was sent from the client application to the `SampleQ1` queue on the message server and then returned to the client as a receiver on that queue. The connection is active.

3. Stop the message server by pressing **Ctrl+C** in the message server window. The connection is broken. The `ReliableTalk` application tries repeatedly to reconnect, as shown in [Figure 11](#).



```
PTP - ..\SonicMQ ReliableTalk -u AlwaysUp -qr SampleQ1 -qs SampleQ1
[ MESSAGE RECEIVED ] AlwaysUp: sending to queue

There is a problem with the connection.
JMSException: Connection dropped
Please wait while the application tries to re-establish the connection...
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
Attempting to create connection...
Cannot connect to broker: localhost:2506. Pausing 10 seconds before retry.
```

Figure 11. ReliableTalk Sample Trying to Reconnect

4. Restart the message server by using its Windows **Start** menu command or the `startbr` script. The `ReliableTalk` application reconnects.

ReliableChat Application (Pub/Sub)

► **To run the ReliableChat sample:**

1. Open a console window to the `TopicPubSub\ReliableChat` folder, then enter:

```
.. \. . \SonicMQ ReliableChat -u AlwaysUp
```
2. Type text and then press **Enter**. The text is displayed, preceded by the user name that initiated that text. The message was sent from the client application to the message server and then returned to the client as a subscriber to that topic. The connection is active.
3. Stop the message server by pressing **Ctrl+C** in the message server window. The connection is broken. The `ReliableChat` application tries repeatedly to reconnect.
4. Restart the message server by using its Windows **Start** menu command or the `startbr` script. The `ReliableChat` application reconnects.

Persistent Storage Application (PTP)

When a message is sent to a queue, the sender can take steps to assure that messages sent are placed on the specified queue with some additional requirements:

- By setting the message delivery mode to **PERSISTENT**, the message is logged before the producer is acknowledged and is guaranteed to be retained in the final message server's message store until it is either acknowledged as delivered or it expires.
- By setting the **JMS_SonicMQ_preserveUndelivered** message property to **true**, if the message is for any reason undelivered, retain it.
- By setting the **JMS_SonicMQ_notifyUndelivered** message property to **true**, send notice to the administrator of the server that manages the queue.

Every message server provides a dead message queue where messages appropriately flagged are moved when they become expired or undeliverable because a destination on that message server or another remote message server puts message delivery into jeopardy.

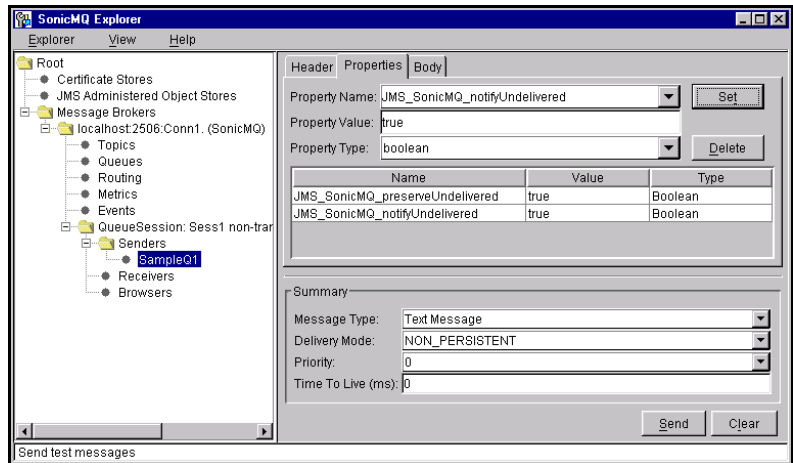
The `DeadMessages` application demonstrates a viewer that looks at the dead message queue. To set up this sample, the Explorer tool is used to create a **PERSISTENT** message that has the properties that will let it become a dead message promptly: a short time to live (expires in a few seconds), and a property setting that indicates that this message should be persisted.

Note Dynamic routing exposes several other reasons a message could get enqueued in the Dead Message Queue. In a variation of this sample, a message could be unexpired yet become undeliverable because it is sent to a bad node (such as `BadNode: : SampleQ1`) or a bad destination (such as `: : BadQ`). See the “Guaranteeing Messages” chapter in the *SonicMQ Deployment Guide* for detailed examples of each reason code.

► **To create a queued message that expires yet persists:**

1. Choose **Start > Programs > Progress SonicMQ > Explorer**. The SonicMQ Explorer window opens at its root level.
2. Click on **Message Brokers** in the Explorer tree. On the right panel:
 - 2.1 Enter Broker Host **local host: 2506**.
 - 2.2 Type **Conn1** as the ConnectID.
 - 2.3 Enter User Name **Administrator** and password **Administrator**.
 - 2.4 Choose **Connect**.
3. In the Explorer tree, click on the message server you just connected to: **local host: 2506: Conn1**.
On the right panel:
 - 3.1 Choose the **Queue** session Type.
 - 3.2 Type **Sess1** for the Name of the new session.
 - 3.3 Click **Create**.
4. In the Explorer tree, click on **Senders**. On the right panel:
 - 4.1 Type **Sample01** as the Queue name.
 - 4.2 Click **Create**.
5. In the Explorer tree, click on the node you just created: **Sender: Sample01**.
On the right panel:
 - 5.1 On the **Body** tab:
 - Enter some text for the body of the message.
 - In the tab's Summary area, choose the Delivery Mode option **PERSISTENT** and then enter a Time To Live value greater than zero, yet brief, say **1000** — 1 second.
 - 5.2 On the **Properties** tab:
 - Choose the Property Name **JMS_SonicMQ_preserveUndelivered**.
 - Enter the Property Value **true** then choose **Set**.

- Choose the Property Name `JMS_SonicMQ_notifyUndelivered`. The Property Value `true` is carried forward then choose `Set`.



5.3 Click **Send**.

The message will be enqueued on `Sample01` for one second. If you had put an active receiver on that queue before the message expired, you would see that the message was listed in `Sample01`, awaiting receivers on that queue. Then you would have taken it off the queue. That would have defeated what we wanted to look at in the sample: A message that expires waiting for a receiver.

Messages that have expired are not removed from the original queue until they are examined by the message server and noted to be expired. The next process uses the queue browser to notice that the message is expired so that it is dequeued from `Sample01` and re-enqueued in the dead message queue, `SonicMQ.deadMessage`.

Note If you do not force expired messages to be reviewed, you can wait for a system refresh to pass over the queues. Two settings in the broker `.ini` file control periodic checks of queues for expired messages:

- `ENABLE_DYNAMIC_QUEUE_CLEANUP=TRUE`
- `QUEUE_CLEANUP_INTERVAL=600` (in seconds, 10 minutes in this example).

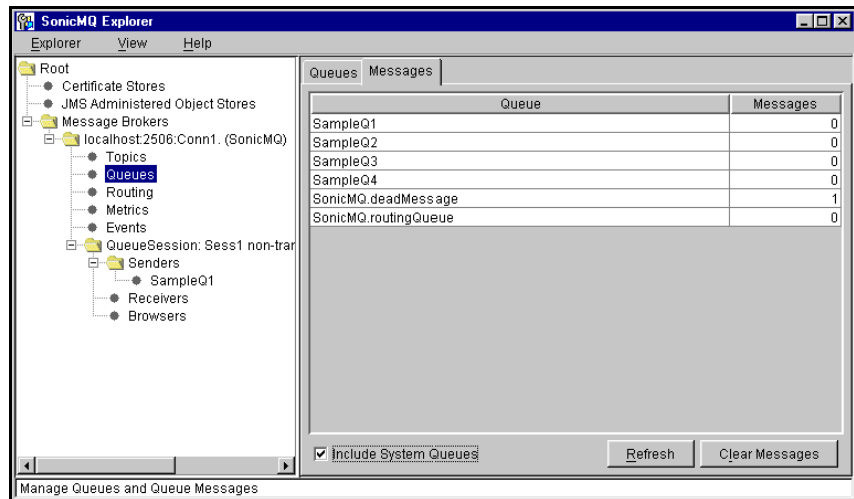
➤ **To browse the message queue to force action on expired messages:**

1. Open a console window to the QueuePTP\QueueMoni tor folder.
2. Type `.. \. . \Soni cMQ QueueMoni tor` and press **Enter**. The QueueMonitor window opens.
3. Click the Browse Queues button. No messages display.

Expired messages were examined and, with the appropriate properties set, are transferred to the dead message queue. The property you set instructs the message server to transfer the expired message to the Dead Message Queue, placing it under administrative control with no expiration. The message must now be explicitly flushed or dequeued.

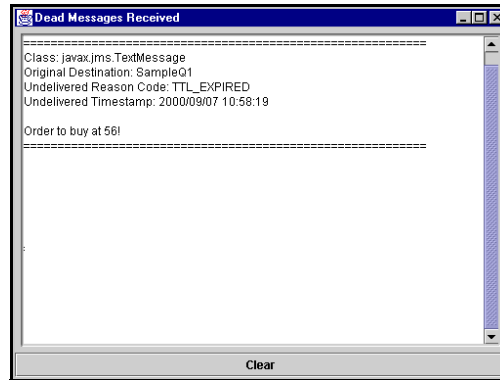
➤ **To see the Explorer view of the Dead Message Queue:**

1. Continuing in the Explorer session where you logged in as Administrator, choose **Queues** in the Explorer tree.
2. Select the **Messages** tab
3. Click the **Refresh** button.
4. Select the option to **Include System Queues**. The dead message queue, `Soni cMQ. deadMessage`, indicates that there is one message enqueued while there are no messages in `Sampl eQ1`.



► **To start the DeadMessages browser sample:**

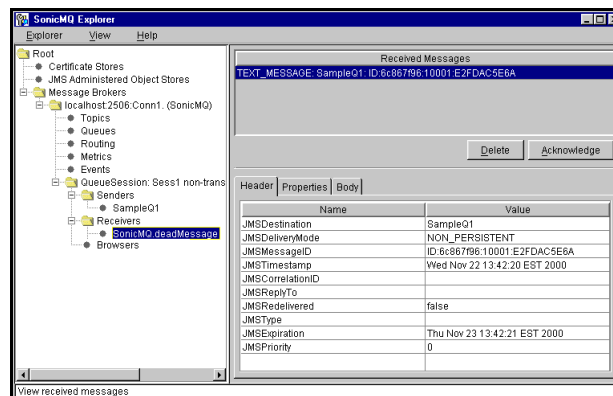
1. Open a console window to the QueuePTP\DeadMessages folder.
2. Type `.. \. \Soni cMQ DeadMessages` and press **Enter**.
3. The dead messages are listed in the DeadMessage browser window as this example shows:



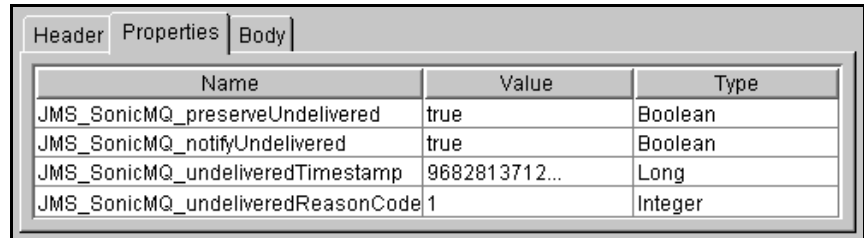
► **To see the contents of the dead message:**

The Explorer lets you look at messages in the Dead Message Queue (DMQ).

1. Open an Explorer session then set up a QueueSession and choose to be a receiver on the system queue, Soni cMQ. deadMessage.
2. Click on the item listed under Received Messages. The header fields display as shown in the following Explorer widow:



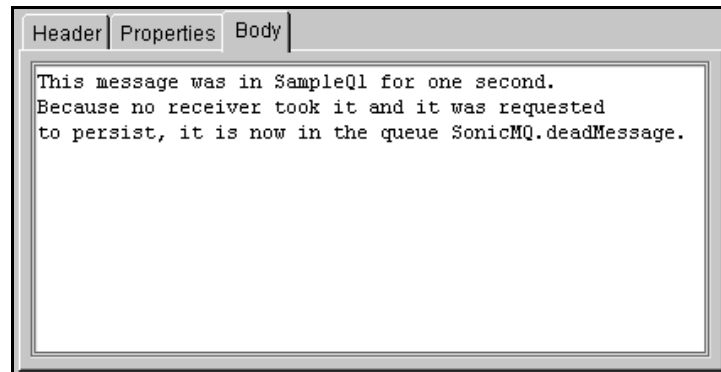
3. Choose the Properties tab. The properties of the undelivered, expired message in the sample are those shown in this window excerpt:



Name	Value	Type
JMS_SonicMQ_preserveUndelivered	true	Boolean
JMS_SonicMQ_notifyUndelivered	true	Boolean
JMS_SonicMQ_undeliveredTimestamp	9682813712...	Long
JMS_SonicMQ_undeliveredReasonCode	1	Integer

The properties carry the original settings to preserve and notify when undelivered. The undelivered timestamp indicates the time of dequeuing into the DMQ. The reason code, 1, indicates that the message expired.

4. Choose the Body tab. The body is unchanged. It might appear like this:



```
This message was in SampleQ1 for one second.
Because no receiver took it and it was requested
to persist, it is now in the queue SonicMQ.deadMessage.
```

A management application might clone the body into a new message to get the message enroute around the reason for unsuccessful delivery.

While expiration is common to all messaging deployments, there are several other reasons a messages could be in doubt or undeliverable in a dynamic routing architecture.

See the *SonicMQ Deployment Guide* for information about the dead message queue and the dynamic routing architecture.

DurableChat Application (Pub/Sub)

When messages are published, they are delivered to all active subscribers. Some subscribers register an enduring interest in receiving messages that were sent while they were inactive. These **durable subscriptions** are permanent records in the message server's database.

Whenever a subscriber connects to the topic again (under the registered username, subscriber name and client identifier) all undelivered messages to that topic that have not expired will be delivered immediately. The administrator can terminate durable subscriptions or a client can use the `unsubscribe` method to close the durable subscription.

In an application, there are only a few changes to set up a subscriber as a durable subscriber. Where Chat was coded as:

```
subscriber = subSession.createSubscriber(topic);
```

DurableChat is:

```
//Durable Subscriptions index on username, clientID, subscription name  
//It is a good practice to set the clientID:  
connection.setClientID(CLIENT_ID);  
subscriber = subSession.createDurableSubscriber  
    (topic, "SampleSubscription");
```

As with ReliableChat, using the **PERSISTENT** delivery mode ensures that messages are logged before they are acknowledged and are non-volatile in the event of a message server failure.

[Figure 12](#) shows what occurs when the subscriber requests an extra effort to ensure delivery.

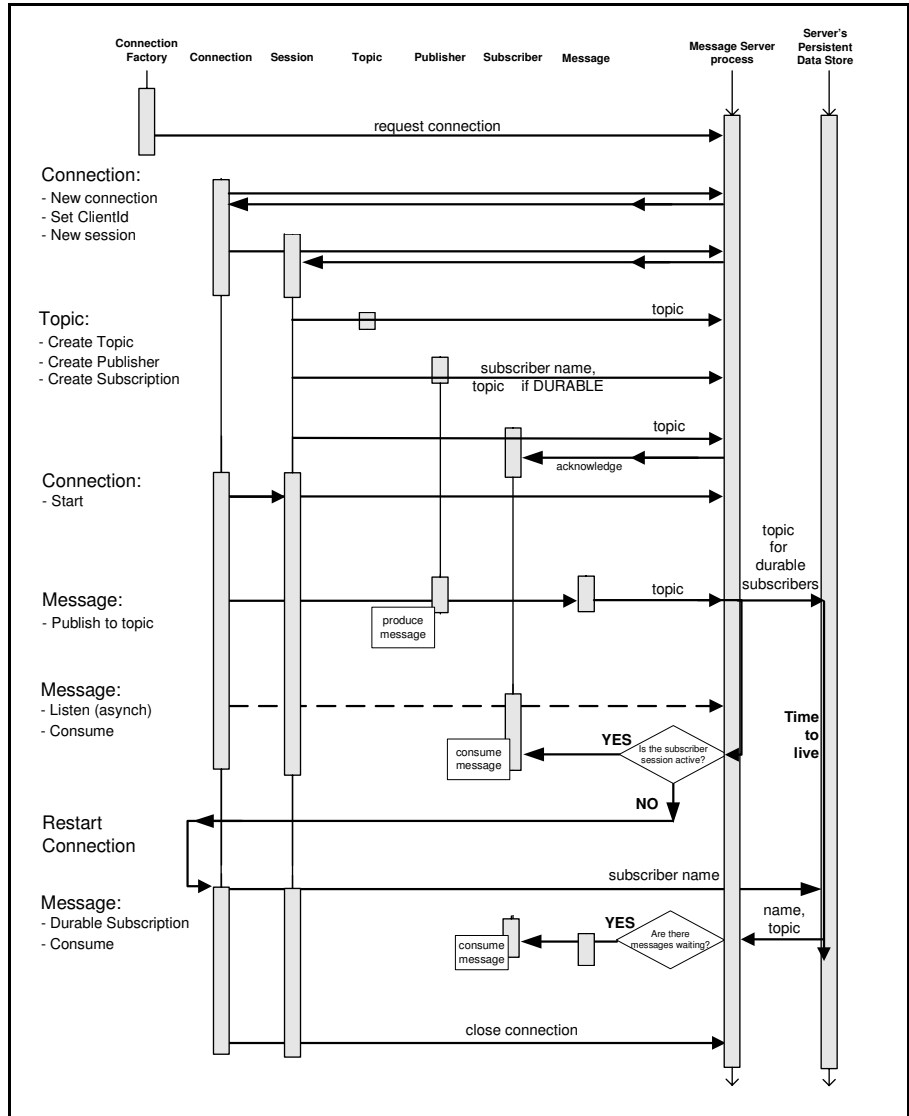


Figure 12. Sequence Diagram for the DurableChat Application

► **To start DurableChat sessions:**

1. Open a console window to the `TopicPubSub\Durabl eChat` folder, then enter:
`.. \. \Soni cMQ Durabl eChat -u AI waysUp`
2. Open another console window to the `TopicPubSub\Durabl eChat` folder, then enter:
`.. \. \Soni cMQ Durabl eChat -u Someti mesDown`
3. In the `AI waysUp` window, type text and then press **Enter**.
The text is displayed on both subscriber's consoles.
4. In the `Someti mesDown` window, type text and then press **Enter**.
The text is displayed on both subscriber's consoles.
5. Stop the `Someti mesDown` session by pressing **Ctrl+C**.
6. In the `AI waysUp` window, send one or more messages.
The text is displayed on that subscriber's console.
7. In the window where you stopped the `Durabl eChat` session, restart the session under the same name.

When the `Durabl eChat` session re-connects, the retained messages are delivered and then displayed.

While durable, the messages were not implicitly everlasting. The publisher of the message sets a **time-to-live** parameter—a value that, when added to the publication timestamp, determines the expiration time of the message. The time-to-live value in milliseconds can be any positive integer. In this sample, the time-to-live is 1,800,000 milliseconds (thirty minutes). Setting the value to zero retains the message indefinitely.

Reviewing Reliable, Persistent, and Durable Messaging

The characteristics that made for a better Quality of Service provide their benefits with modest overhead. The examples in this section can be combined so that you create a reliable, persistent talk and a reliable, durable chat. The source code of these samples can provide snippets that are readily transferable into your applications.

There are many other facets to optimal QoS, including the various security, encryption, access control, and transport protocols. For more information, see the *SonicMQ Installation and Administration Guide*.

Request and Reply Samples

The advantages of loosely coupled applications call for special techniques when it is important for the publisher to certify that a message was delivered in either messaging domain:

- **Point-to-Point** — While a sender can see if a message was removed from a queue, implying that it was delivered, there is no indication where it went.
- **Publish and Subscribe** — While the publisher can send long-lived messages to durable receivers and get acknowledgement from the message server, neither of these techniques confirms that a message was actually delivered or how many, if any, subscribers received the message.

A message producer can request a reply when a message is sent. A common way to do this is to set up a **temporary destination** and header information that the consumer can use to create a reply to the sender of the original message.

In both Request and Reply samples, the replier's task is a simple data processing exercise: standardize the case of the text sent—receive text and send back the same text as either all uppercase characters or all lowercase characters—then publish the modified message to the temporary destination that was set up for the reply.

While request-and-reply provides proof of delivery, it is a blocking transaction—the requestor waits until the reply arrives. While this situation might be appropriate for a system that, for example, issues lottery tickets, it might be preferable in other situations to have a formally established return destination that echoes the original message and a **correlation identifier**—a designated identifier that certifies that each reply is referred to its original requestor.

Note `JMSReplyTo` and `JMSCorrelationID` are used as a suggested design pattern established as a part of the JMS specification. The application programmer ultimately decides how these fields are used, if they are used at all.

The sample applications use JMS sample classes, `TopicRequestor` and `QueueRequestor`. You should create the Request/Reply helper classes that are appropriate for your application.

Request and Reply (PTP)

In the PTP domain, the requestor application can be started and even send a message before the replier application is started. The queue holds the message until the replier is available. The requestor is still blocked, but when the replier's message listener receives the message, it releases the blocked requestor. The sample code includes an option (-m) to switch the mode between upper and lowercase.

➤ **To set up the PTP Request Reply sessions:**

- Open two console windows to the QueuePTP\RequestReply folder.

➤ **To start the PTP Requestor session:**

- In one console window, enter:
`.. \. . \SonicMQ Requestor -u QREQUESTOR`

➤ **To start the PTP Replier session:**

- In the other console window, enter:
`.. \. . \SonicMQ Replier -u QReplier`
The default value of the mode is uppercase.

➤ **To test a PTP request and reply:**

- In the Requestor window, type `AaBbCc` and then press **Enter**.

The Replier window reflects the activity, displaying:

```
[Request] QRequestor: AaBbCc
```

The Replier does its operation (converts text to uppercase) and sends the result in a message to the Requestor. The Requestor window gets the reply from the Replier:

```
[Reply] Uppercasing-QRequestor: AABbcc
```

Request and Reply (Pub/Sub)

➤ **To set up the Pub/Sub RequestReply sessions:**

- ❑ Open two console windows to the `TopicPubSub\RequestReply` folder.

➤ **To start the Pub/Sub Replier session:**

Important Start the replier before the requestor so that the Pub/Sub replier's message listener can receive the message and release the blocked requestor.

- ❑ In one of the windows, type `.. \. \SonicMQ Replier` and press **Enter**. The default value for mode will be used, uppercase.

➤ **To start the Pub/Sub Requestor session:**

- ❑ In the other window, type `.. \. \SonicMQ Requestor` and press **Enter**.

➤ **To test a Pub/Sub request and reply:**

- ❑ In the **Requestor** window, type `AaBbCc` and then press **Enter**.

The **Replier** window reflects the activity, displaying:

```
[Request] SampleReplier: AaBbCc
```

The replier does its operation (convert text to uppercase) and sends the result in a message to the requestor. The requestor gets the reply from the replier:

```
[Reply] Uppercasing-SAMPLEREQUESTOR: AABBCc
```

Reviewing the Request and Reply Samples

In review, these samples show:

- Request and reply mechanisms are very similar across domains.
- While there might be zero or many subscriber replies, there will be, at most, one PTP reply.
- Using message header fields (`JMSReplyTo` and `JMSCorrelationID`) and the requestor sample classes (`javax.jms.TopicRequestor` and `javax.jms.QueueRequestor`) are suggested implementations for request-and-reply behavior in JMS.

Selection and Wildcard Samples

While specific queues and topics provide focused content nodes for messages that are of interest to an application, there are circumstances when the programmer may want to qualify the scope of interest a consumer has in messages much like a SQL WHERE clause.

Conversely there are circumstances where the specificity of having to declare each topic of interest becomes slow and unwieldy. Because topic names can be created as needed (assuming there are no security constraints), a subscriber application may need to scan many topics.

These situations are contrasted in these samples:

- If you force too much traffic into a small number of destinations and then use selector strings, performance takes a substantial hit in most deployments.
- If you use a lot of topic names, SonicMQ's hierarchical topic structure bypasses a lot of message selector overhead. The ability to apply wildcards to subscriptions can provide oversight by just subscribing to parent topic nodes.

SelectorTalk Application (PTP)

In the SelectorTalk application, the application starts by declaring a selector *String-value* that will be attached to the message as `PROPERTY_NAME='String-value'`. The send and receive to alternate queues so that they pass each other messages. The receive method has a selector string parameter (-s). In PTP domains, all messages for a queue topic are filtered on the message server and then the qualified messages are delivered to the consumer.

► **To start SelectorTalk sessions:**

1. Open a console window to the QueuePTP\SelectorTalk folder, then enter
`.. \. \SonicMQ SelectorTalk -u AAA -s North -qr SampleQ1 -qs SampleQ2`
2. Open another console window to the QueuePTP\SelectorTalk folder, then enter:
`.. \. \SonicMQ SelectorTalk -u BBB -s South -qr SampleQ2 -qs SampleQ1`

► To SelectorTalk:

1. In the **AAA** window, type any text and then press **Enter**. The message is enqueued but there is no receiver. The **BBB** selector string does not see any enqueued messages except those that evaluate to **South**.
2. Stop the **BBB** session by pressing **Ctrl+C**.
3. In that console window start a new session, changing the selector string:

```
.. \. \SonicMQ SelectorTalk -u BBB -s North -qr SampleQ2 -qs Sample q1
```

The session starts and the message that was enqueued is immediately received.
4. In the **AAA** window, again type any text and then press **Enter**. The message is enqueued and the **BBB** selector string qualifies the message for delivery.

SelectorChat Application (Pub/Sub)

In the `SelectorChat` application, the application starts by declaring the *String-value* that will be attached to the message as `PROPERTY_NAME='String-value'`. The method for the subscription declares the sample's topic, `jms.samples.chat`, and the selector string (`-s`). In Pub/Sub domains, all messages for a subscribed topic are delivered to the subscriber and then the filter is applied to decide what will be consumed.

► To start SelectorChat sessions:

1. Open a console window to the `TopicPubSub\SelectorChat` folder, then enter:

```
.. \. \SonicMQ SelectorChat -u Closer -s Sales
```
2. Open another console window to the `TopicPubSub\SelectorChat` folder, then enter:

```
.. \. \SonicMQ SelectorChat -u Presenter -s Marketing
```

► To SelectorChat:

1. In the **Closer** window, type any text and then press **Enter**. The text is only displayed in that window. The **Presenter** selector string excludes the **Sales** message.

2. In the **Presenter** window, type any text and then press **Enter**. The text is only displayed in that window. The **Closer** selector string excludes the **Marketing** message.
3. Stop the **Closer** session by pressing **Ctrl+C**.
4. In that console window start a new session, changing the selector string:


```
.. \.. \SonicMQ SelectorChat -u Closer -s Marketing
```
5. Type text in either window and then press **Enter**. The text is displayed in both windows. The selector string matches and the message displays.

Hierarchical Chat Application (Pub/Sub)

SonicMQ lets an application have the power of a message selector plus a more streamlined way to often get the same result: A hierarchical topic structure that allows wildcard subscriptions. In this sample, each application instance creates two sessions, one for the publish topic (**-t**) and one for the subscribe topic (**-s**).

► To start HierarchicalChat sessions:

1. Open a console window to the `TopicPubSub\HierarchicalChat` folder then type


```
.. \.. \SonicMQ HierarchicalChat -u HQ -t sales.corp -s sales.*
```

 and press **Enter**.
2. Open another console window to the `TopicPubSub\HierarchicalChat` folder.
3. Type


```
.. \.. \SonicMQ HierarchicalChat -u America -t sales.usa -s sales.usa
```

 and press **Enter**.

► To HierarchicalChat:

1. In the **HQ** window, type text and then press **Enter**. The text is displayed in only the **HQ** window because it subscribes to all topics in the sales hierarchy and **America** is subscribing to only the `sales.usa` topic.
2. In the **America** window, type text and then press **Enter**. The text is displayed in both windows:
 - **America** subscribes to the `sales.usa` topic.
 - **HQ** subscribes to all topics that start with “`sales.`”.

Reviewing the Selection and Wildcard Samples

While selector strings can provide a variety of ways to qualify what messages will be carried to a consumer, the overhead in the evaluation of the selectors can slow down overall system performance. Hierarchical Chat illustrates a feature of SonicMQ that can provide the advantages of selectors with minimal overhead. Note also that security access control uses similar wildcard techniques to enable read/write security for all subtopics within a topic node.

Another way to increase specificity is to use complex SQL statements. For information on hierarchical security, including hierarchical name spaces and security, see the *SonicMQ Installation and Administration Guide*.

Test Loop Sample

A simple loop test lets you experiment with messaging performance.

QueueRoundTrip Application (PTP)

The RoundTrip application sends a brief message to a sample queue and then uses a temporary queue to receive the message back. A counter is incremented and the message is sent for another trip. After completing the number of cycles you entered when you started the test, the run completes by displaying summary and average statistics.

► **To run QueueRoundTrip:**

1. Open a console window to the QueuePTP\QueueRoundTrip folder.
2. Enter:

```
.. \. \SonicMQ QueueRoundTrip -n 100
```
3. Look at the results.
4. Enter:

```
.. \. \SonicMQ QueueRoundTrip -n 1000
```

Note This sample lets you evaluate features and is not intended as a performance tool. For information on performance, see the *SonicMQ Deployment Guide*.

Extending the Samples

After exploring the samples you can modify the sample source files to learn more about SonicMQ. You need a Java compiler to compile your changes.

Use Common Topics Across Clients

When you run the Pub/Sub samples you might notice that while all the Chat applications get Chat messages and all the DurableChat applications get DurableChat messages, they do not receive each other's messages. This is because the applications are publishing to different topics. You can set the two applications to monitor messages on the same topic.

► **To put Chat and DurableChat on the same topic:**

1. Edit the SonicMQ sample file `DurableChat.java`.
2. Change the variable `APP_TOPIC` from `jms.samples.durablechat` to `jms.samples.chat`.
3. Save and compile the edited `.java` file.
4. Run the edited `.class` file.

Now messages sent from DurableChat and Chat are received by both regular and durable subscribers. The durable subscribers will receive messages when they recover from offline situations, but the regular subscribers will not recover missed messages.

Important If you make this change, the message server will maintain the durable subscriptions for all the Chat messages. While DurableChat messages expire after 30 minutes, Chat messages are published with the default time-to-live (never expire). The Chat messages will endure for durable subscribers until one of the following occurs:

- The durable subscriber connects to receive the messages.
- The durable subscriber explicitly unsubscribes.
- The database is initialized.

Trying Different RoundTrip Settings

The RoundTrip application lets you choose a number of produce-then-consume iterations to perform when the application runs. You can enhance the application to explore the time impact of other settings and parameters as well.

Note This sample lets you evaluate features and is not intended as a performance tool. For information on performance, see the *SonicMQ Deployment Guide*.

A counter is incremented and the message is sent for another trip. After completing the number of cycles you entered when you started the test, the run completes by displaying summary and average statistics.

► **To extend the QueueRoundTrip sample:**

1. Edit the SonicMQ sample file `QueuePTP\QueueRoundTrip.java` to establish any of the following behavior changes:
 - Change the `javax.jms.message.DeliveryMode` from `NON_PERSISTENT` to `PERSISTENT`. Run it then change it to `NON_PERSISTENT_ASYNC`.
 - You could change the `priority` or `timeToLive` values but in this sample the effect would be negligible.
 - Change the message type from the `bodyless createMessage()` to a bodied message type, such as `createTextMessage()`.
 - Create a set of sample strings (or other appropriate data type) to populate a bodied-message payload with different size payloads.
 - Use `createXMLMessage()` and load the message payload with well-formed XML data. Then try the same payload as a `TextMessage`.
 - Change the receiver session acknowledgement mode from `AUTO_ACKNOWLEDGE` to `DUPS_OK_ACKNOWLEDGEMENT`. Change it again to `CLIENT_ACKNOWLEDGE` or `SINGLE_MESSAGE_ACKNOWLEDGE` then add an explicit `acknowledge()` after the receive is completed.
2. Save and compile the edited `.java` file.
3. Open a console window to the `QueuePTP\QueueRoundTrip` folder.
4. Type `..\..\SonicMQ\QueueRoundTrip -n 100`
5. Look at the results and compare them to other round trips.

Modifying the MapMessage to Use Other Data Types

The concept of the MapMessage sample is limited when its content is just a snippet of text. The key concepts of the MapMessage are that:

- The body is a collection of name-value pairs.
- The values can be Java primitives.
- The receiver can access the names in any sequence.
- The receiver can attempt to coerce a value to another data type.

The following exercise adds some mixed data types to the MapTalk source file before the message is sent. Then the receiver takes the data in a different sequence and formats it for display.

The example uses typed `set` methods to populate the message with name-typedValue pairs. The `get` methods retrieve the named properties and attempt coercion if the data type is dissimilar.

► To extend the MapTalk sample to use and display other data types:

1. Edit the SonicMQ sample file `MapTalk.java` at the lines:

```
javax.jms.MapMessage msg = sendSession.createMapMessage();
msg.setString("sender", username);
msg.setString("content", s);
```

2. Add the lines for the `set` methods (or your similar lines):

```
msg.setInt("FiscalYearEnd", 10);
msg.setString("Distribution", "global");
msg.setBoolean("LineOfCredit", true);
```

3. You must extract the additional data by `get` methods to expose the values in the receiving application. Because the sample is a text-based display, you can include the `getString` methods in the construction of the string that will display in the console.

Change:

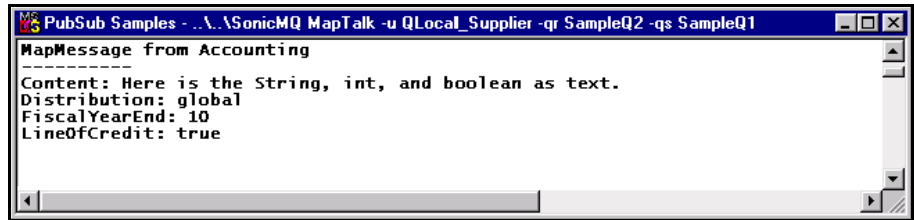
```
String content = mapMessage.getString("content");
System.out.println(sender + ": " + content);
```

to:

```
String content =
    ("Content: " + mapMessage.getString("content") + "\n" +
     "Distribution: " + mapMessage.getString("Distribution") + "\n" +
     "FiscalYearEnd: " + mapMessage.getString("FiscalYearEnd") + "\n" +
     "LineOfCredit: " + mapMessage.getString("LineOfCredit") + "\n");
System.out.println("MapMessage from " + sender +
                  "\n----- \n" + content);
```

4. Save and compile the edited .java file.
5. Run the edited .class file.

Now when the MapTalk sample runs, the content is the text you typed plus the mapped, resequenced, and converted map properties.



Modifying the XMLMessage to Show More Data

The sample for the `XMLMessage` type is limited to the data that is input as text as a single content node. While the data collection/validation loops and the data transfers from application data stores are reserved as more advanced exercises, this example demonstrates how well-formed XML data is transformed into `DocNodes` from the `org.w3c.dom` Node standards.

► **To extend the XMLChat sample to show more data:**

1. Edit the SonicMQ sample file XMLChat.java starting after:

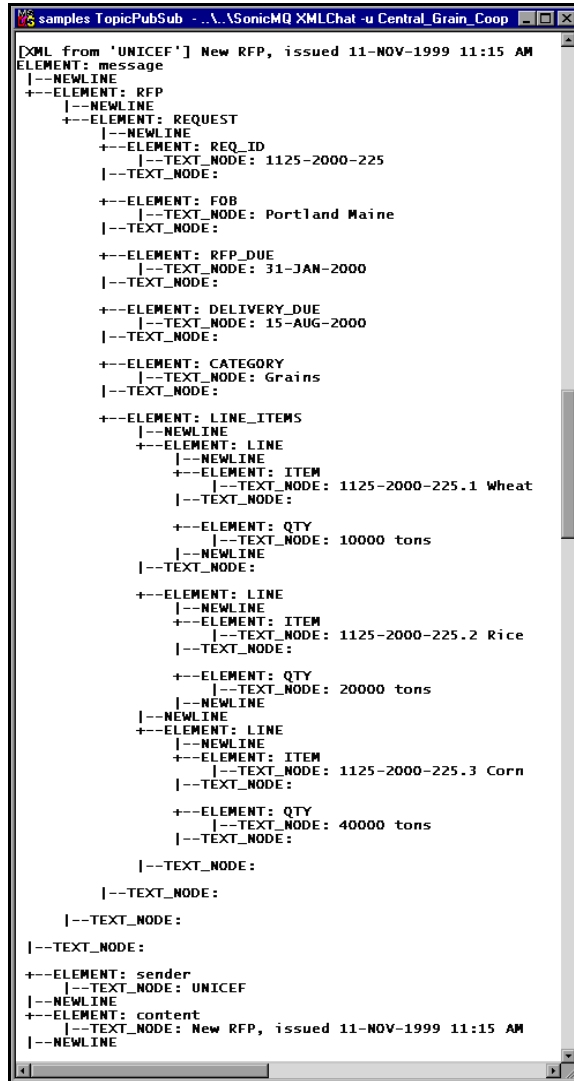
```
// Note that the XMLMessage is a Progress Software extension
progress.message.client.XMLMessage xMsg =
...
StringBuffer msg = new StringBuffer();
msg.append("<?xml version='1.0'?>\n");
msg.append("<message>\n");
msg.append("  <sender> + username + "</sender>\n");
msg.append("  <content> + s + "</content>\n");
```

2. Insert the formatted, tagged XML lines you want to append to the message, for example:

```
msg.append("<RFP>\n");
msg.append("<REQUEST>\n");
msg.append("<REQ_ID>1125-2000-225</REQ_ID> \n");
msg.append("<FOB>Portland Maine</FOB> \n");
msg.append("<RFP_DUE>31-JAN-2000</RFP_DUE> \n");
msg.append("<DELIVERY_DUE>15-AUG-2000</DELIVERY_DUE> \n");
msg.append("<CATEGORY>Grains</CATEGORY> \n");
msg.append("<LINE_ITEMS>\n");
msg.append("<LINE>\n");
msg.append("<ITEM>1125-2000-225.1 Wheat</ITEM> \n");
msg.append("<QTY>10000 tons</QTY>\n");
msg.append("</LINE> \n");
msg.append("<LINE>\n");
msg.append("<ITEM>1125-2000-225.2 Rice</ITEM> \n");
msg.append("<QTY>20000 tons</QTY>\n");
msg.append("</LINE>\n");
msg.append("<LINE>\n");
msg.append("<ITEM>1125-2000-225.3 Corn</ITEM> \n");
msg.append("<QTY>40000 tons</QTY> \n");
msg.append("</LINE> \n");
msg.append("</LINE_ITEMS> \n");
msg.append("</REQUEST> \n");
msg.append("</RFP> \n");
msg.append("</message> \n");
```

3. Save and compile the edited .java file.
4. Run the edited .class file.

When you run the application and enter a basic text message, the complete document object model (DOM) is also displayed, similar to the subscriber session listing in [Figure 13](#).



```
[XML from 'UNICEF'] New RFP, issued 11-NOV-1999 11:15 AM
ELEMENT: message
|--NEWLINE
+--ELEMENT: RFP
|--NEWLINE
+--ELEMENT: REQUEST
|--NEWLINE
+--ELEMENT: REQ_ID
|--TEXT_NODE: 1125-2000-225
|--TEXT_NODE:
+--ELEMENT: FOB
|--TEXT_NODE: Portland Maine
|--TEXT_NODE:
+--ELEMENT: RFP_DUE
|--TEXT_NODE: 31-JAN-2000
|--TEXT_NODE:
+--ELEMENT: DELIVERY_DUE
|--TEXT_NODE: 15-AUG-2000
|--TEXT_NODE:
+--ELEMENT: CATEGORY
|--TEXT_NODE: Grains
|--TEXT_NODE:
+--ELEMENT: LINE_ITEMS
|--NEWLINE
+--ELEMENT: LINE
|--NEWLINE
+--ELEMENT: ITEM
|--TEXT_NODE: 1125-2000-225.1 Wheat
|--TEXT_NODE:
+--ELEMENT: QTY
|--TEXT_NODE: 10000 tons
|--NEWLINE
|--TEXT_NODE:
+--ELEMENT: LINE
|--NEWLINE
+--ELEMENT: ITEM
|--TEXT_NODE: 1125-2000-225.2 Rice
|--TEXT_NODE:
+--ELEMENT: QTY
|--TEXT_NODE: 20000 tons
|--NEWLINE
+--ELEMENT: LINE
|--NEWLINE
+--ELEMENT: ITEM
|--TEXT_NODE: 1125-2000-225.3 Corn
|--TEXT_NODE:
+--ELEMENT: QTY
|--TEXT_NODE: 40000 tons
|--TEXT_NODE:
|--TEXT_NODE:
|--TEXT_NODE:
|--TEXT_NODE:
|--TEXT_NODE:
+--ELEMENT: sender
|--TEXT_NODE: UNICEF
|--NEWLINE
+--ELEMENT: content
|--TEXT_NODE: New RFP, issued 11-NOV-1999 11:15 AM
|--NEWLINE
```

Figure 13. XMLMessage Parsed into a Document Object Model

As the data is interpreted in the DOM format only when the message is an instance of an `XMLMessage`, a Chat session would display the same message as a `TextMessage`—the XML-tagged text without DOM interpretation, as shown in Figure 14.

```

<?xml version="1.0" ?>
<message>
<RFP>
<REQUEST>
<REQ_ID>1125-2000-225</REQ_ID>
<FOB>Portland Maine</FOB>
<RFP_DUE>31-JAN-2000</RFP_DUE>
<DELIVERY_DUE>15-AUG-2000</DELIVERY_DUE>
<CATEGORY>Grains</CATEGORY>
<LINE_ITEMS>
<LINE>
<ITEM>1125-2000-225.1 Wheat</ITEM>
<QTY>10000 tons</QTY>
</LINE>
<LINE>
<ITEM>1125-2000-225.2 Rice</ITEM>
<QTY>20000 tons</QTY>
</LINE>
<LINE>
<ITEM>1125-2000-225.3 Corn</ITEM>
<QTY>40000 tons</QTY>
</LINE>
</LINE_ITEMS>
</REQUEST>
</RFP>
  <sender>UNICEF</sender>
  <content>New RFP, issued 11-NOV-1999 11:15 AM</content>
</message>
    
```

Figure 14. XMLMessage as Tagged Text

Note You could have appended the XML tagged lines without the `\n`. That would suppress the blank `TEXT_MODE` lines in the DOM. It would however make one unbroken text line for general text or raw XML review.

Using Samples with Security Initialized

The sample database can be set up with security so that users can be authorized and authenticated for both general access to the message server yet also for permissions to read from and write to destinations.

Warning All data that you have previously put into the database will be lost. See the *SonicMQ Installation and Administration Guide* for information about enabling security, starting administrator tools, and adding users.

► **To set up the security database under Windows:**

1. Close all active clients and then stop the message server.
2. Edit the file `broker.ini` at the root level of your SonicMQ installation to modify the variable `ENABLE_SECURITY` from `FALSE` to `TRUE`.
3. Open a console window to the SonicMQ install directory.
4. Type `bin\dbtool /d b s` and press **Enter**.
5. Type `bin\dbtool /cs basi c` and press **Enter**.
6. Type `bin\dbtool /c securi ty` and press **Enter**.
7. Start the message server again. Notice that security is implemented.
8. Try the Chat sample with the sample name `OTC_Ticker`. It is refused because the only default user in a new security database is `Administrator`.

► **To set up the security database under UNIX:**

1. Close all active clients and then stop the message server.
2. Edit the file `broker.ini` at the root level of your SonicMQ installation to modify the variable `ENABLE_SECURITY` from `FALSE` to `TRUE`.
3. Open a console window to the SonicMQ install directory.
4. Type `./bin/dbtool -d b s` and press **Enter**.
5. Type `./bin/dbtool -cs basi c` and press **Enter**.
6. Type `./bin/dbtool -c securi ty` and press **Enter**.
7. Start the message server again. Notice that security is implemented.
8. Try the Chat sample with the sample name `OTC_Ticker`. It is refused because the only default user in a new security database is `Administrator`.

With security implemented, only user names in the security database can access the message server. Only the Administrator can maintain user records.

► **To set up Users with the Explorer**

1. In the SonicMQ Explorer, choose **Message Brokers**.
2. Enter the Broker Host, typically `local host: 2506`.
3. Type any text as the ConnectID.
4. Type **Administrator** as the User and again as the Password.
5. Click **Connect**.
6. Choose **Users**.
7. For each user you want to set up:
 - 7.1 In the User Maintenance area, click **New**.
 - 7.2 Type the user name, for example `OTC_Ticker`.
 - 7.3 Type a password—for example, `OTC`—then confirm it.
 - 7.4 Click **Update**.

► **To access a secured message server as an authentic user:**

1. Try the Chat sample with a username you set up.
2. Append the password parameter and the selected user's password.

For example, the command to start Chat before using security is:

```
.. \.. \SonicMQ Chat -u OTC_Ticker
```

The command to start Chat under security is:

```
.. \.. \SonicMQ Chat -u OTC_Ticker -p OTC
```

The authenticated user is accepted and the application starts.

When you have finished evaluating security you can resume working with the sample applications, but you will have to:

- Enter every user name into the security database.
- Assign a password to each user.
- Use the password parameter and the password on every application.

Removing Security from the Database

After exploring the security database with the sample applications, you can re-initialize the database to eliminate the security database.

► **To set up the database without security under Windows:**

1. Close all active clients and stop the message server.
2. Edit the file `broker.ini` at the root level of your SonicMQ installation to modify the variable `ENABLE_SECURITY` from `TRUE` to `FALSE`.
3. Open a console window to the SonicMQ install directory.
4. Type `bin\dbtool /d b s` and press **Enter**. The basic tables and security tables are deleted.
5. Type `bin\dbtool /cs basi c` and press **Enter**. The basic tables are created with the sample queues.

► **To set up the database without security under UNIX:**

1. Close all active clients and stop the message server.
2. Edit the file `broker.ini` at the root level of your SonicMQ installation to modify the variable `ENABLE_SECURITY` from `TRUE` to `FALSE`.
3. Open a console window to the SonicMQ install directory.
4. Type `./bin/dbtool -d b s` and press **Enter**. The basic tables and security tables are deleted.
5. Type `./bin/dbtool -cs basi c` and press **Enter**. The basic tables are created with the sample queues.

When you restart the message server, notice that security is disabled.

See the *SonicMQ Installation and Administration Guide* for more information about security and the **dbtool**.

About Client Sessions

The SonicMQ Java client provides a lightweight, 100% Java platform that can access the messaging features provided by the SonicMQ message servers. In the JMS programming model, a programmer creates JMS connections that establish the application's identity and specify how the connection with the message server will be maintained. Within each connection, one or more sessions are established. Each session is used for a unique delivery thread for messages that are delivered to the client application. This chapter presents the programming required to establish and maintain client connections to message servers through sessions.

Identifiers

SonicMQ uses several identifiers to differentiate and distinguish application registrations. The following information describes the primary identifiers—`connectID`, `username`, `clientID`, and `subscriptionName`—and how they are used in messaging.

ConnectID

The `ConnectID` is a SonicMQ identifier that can control whether the message server allows multiple connections for a user in a client application:

- To assure that a connection for a user name is exclusive such that no other connection can use that `username/ConnectID` combination until the

connection is closed, use the appropriate set method and pass a non-null string:

- `TopicConnectionFactory.setConnectID(String connectid)`
- `QueueConnectionFactory.setConnectID(String connectid)`
- To allow other connections for a username/ConnectID, use the appropriate set method and pass the null string as the parameter:
 - `TopicConnectionFactory.setConnectID("")`
 - `QueueConnectionFactory.setConnectID("")`

ConnectID can also be configured as part of a ConnectionFactory, or passed as an argument to a SonicMQ Connection object constructor.

User Name

A user name (and password) defines a principal's identity maintained by the SonicMQ message server's security database to authenticate a user with the SonicMQ message server and establish privileges and access rights.

When security is not enabled, the user name is simply a text label.

A user name can be:

- Configured in a Connection Factory or passed as a parameter to the constructors:
 - `progress.message.jclient.TopicConnectionFactory`
 - `progress.message.jclient.QueueConnectionFactory`
- Passed as a parameter to the methods:
 - `createTopicConnection(username, password)` of the `TopicConnectionFactory` object
 - `createQueueConnection(username, password)` of the `QueueConnectionFactory` object

ClientID

The `ClientID` is a unique identifier that can avoid conflicts for durable subscriptions when many clients might be using the same user name and the same subscription name.

To set the value of the `ClientID`, do one of the following:

- In the client application, programmatically set the client identifier immediately after creating a connection, using:

```
Connection.setClientID(String clientId)
```

- In a connection factory, pre-configure the client identifier by either:
 - Using the SonicMQ Explorer or Admin tool to configure the ClientID.
 - In the client application, use the appropriate set method:
 - TopicConnectionFactory.setClientID(String clientId)
 - QueueConnectionFactory.setClientID(String clientId)

Note If the connection factory has configured the client identifier, an attempt to `setClientID()` programmatically on the connection throws an `IllegalStateException`.

Subscription Name

A subscription name always includes the name of the topic. To distinguish different message selectors used in subscriptions, you can include a string which helps identify the message selector. For example, you can use a subscription named `Atlas_prioity4` for a subscription to the `Atlas` topic with selector `JMSprioity=4`. This construct lets you create many durable subscriptions that are easily understood and non-conflicting.

The durable subscription identity is then constructed from and indexed on:

- `username` — The user name used for log on authorization or identity
- `clientID` — The instance identifier in an application
- `subscription name` — The identity of the selection criteria in the subscription

Communication Layer

The SonicMQ message server works in concert with the network layer to provide asynchronous message communications between client applications. As shown in [Figure 15](#), a client can send and receive messages through the SonicMQ API and interfaces to communicate on network connection to a message server. Messages might be stored in a message store as an optional service specified by the message producer.

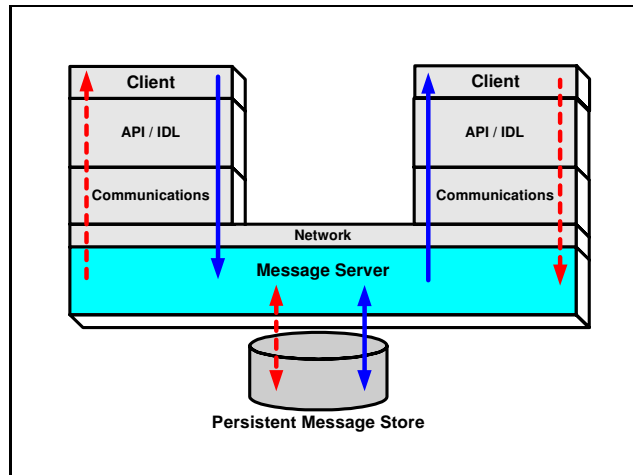


Figure 15. Client - Message Server - Client Communications

The connection layer, as shown in [Figure 16](#), involves getting a connection factory, then creating connections, and finally creating sessions.

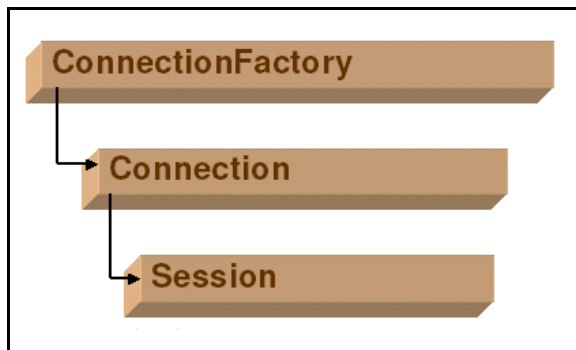


Figure 16. Sessions in Connections from Connection Factories

Each instance of a `ConnectionFactory` is dedicated to only one of the messaging models:

- **Point-to-Point (PTP)** — Messaging is *one-to-one* because only one receiver will get the message. Messages are placed on queues where they endure until a receiver takes delivery and acknowledges receipt.
- **Publish and Subscribe (Pub/Sub)** — Messaging is *one-to-many* or *broadcast* because there could be *zero-to-many* subscribers for a given topic who will each receive the one message that was sent. If no subscribers expressed an enduring interest in a message topic, a message is discarded.

ConnectionFactory

The normal mechanism for establishing a Java connection expects a Java client to create a **ConnectionFactory** with a message server and request authorization to establish a connection for messaging with the behaviors of one of the messaging models. The `TopicConnectionFactory` and `QueueConnectionFactory` are administered objects that encapsulate a set of configuration parameters. The parameters can be assembled at the moment when the application wants to instantiate the object or stored by an administrator in an object store for later recall.

See the *SonicMQ Installation and Administration Guide* for information about administering a `ConnectionFactory`.

The optional mechanisms for creating a `ConnectionFactory` are discussed in detail in the following sections.

Lookup a Stored Context

When an administrator stores the connection parameters as a context, applications can simply access the currently stored values for the named context to connect in a predictable way. The SonicMQ Explorer and Admin tool can explicitly set up a `ConnectionFactory` configuration. The resulting `ConnectionFactory` is then stored as an administered object in a simple file store and then referenced indirectly (by name) to provide the context.

Lookup a Serialized Object in a File Store

Figure 17 diagrams the lookup in a file store at a specified path location. An administrator stores serialized Java objects as flat files with `sj o` extensions. The files can then be retrieved with a `ConnectionFactory` context object reference.

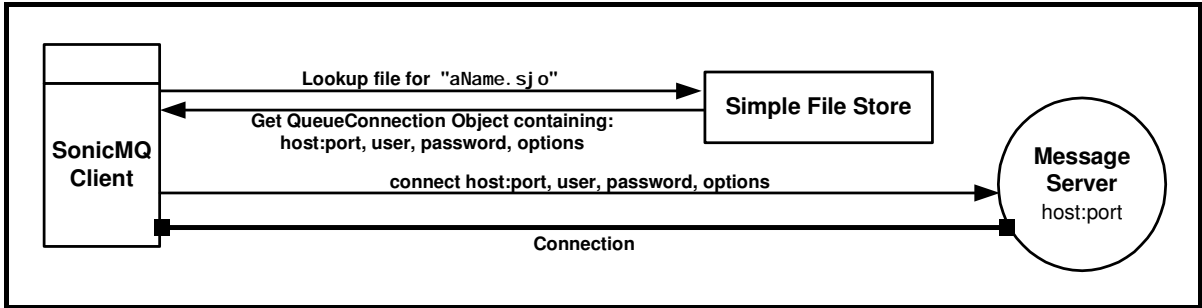


Figure 17. ConnectionFactory Object Instantiated By Lookup of a Serialized Java Object

Lookup on a JNDI LDAP Server

A dispersed system might prefer to use a directory server such as LDAP with lookup through JNDI interfaces. The advantage to this technology is that server records can be maintained remotely and accessed by widely dispersed applications.

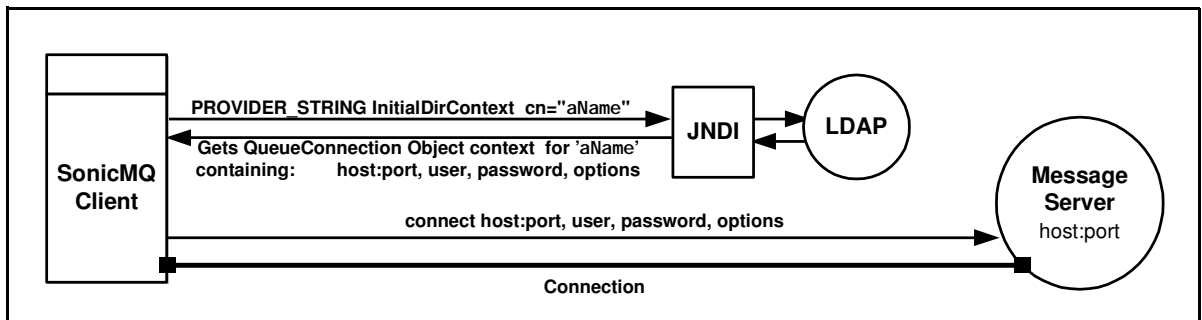


Figure 18. Alternate Connection Techniques Using Factory Objects or JNDI Lookup

This type of lookup submits a value to the LDAP provider with an indexed context name. In Figure 18 the context name `aName` is submitted as `cn=aName` to the server.

The lookup requests the initial context factory from a specified `PROVIDER_STRING` such as:

```
"ldap://mypc.a.sonicmq.com:389/ou=jsao,ou=sonicmq,o=a.sonicmq.com"
```

The JNDI interfaces in `javax.naming` provide directory. `InitialDirContext` `getContext()` for the initial context factory under `com.sun.jndi.ldap.LdapCtxFactory` which then returns the current context values. These are then submitted to the indicated message server where the appropriate `ConnectionFactory` is created.

See [Chapter 12, "Lookup of Administered Objects,"](#) for more information.

Direct Creation of the ConnectionFactory Object

An application can use the SonicMQ API `new` constructor to create a `ConnectionFactory` object. This method usually hard wires many default values into the compiled application, expecting that any overrides to the settings will be read in through a properties file or command-line options when the application is started.

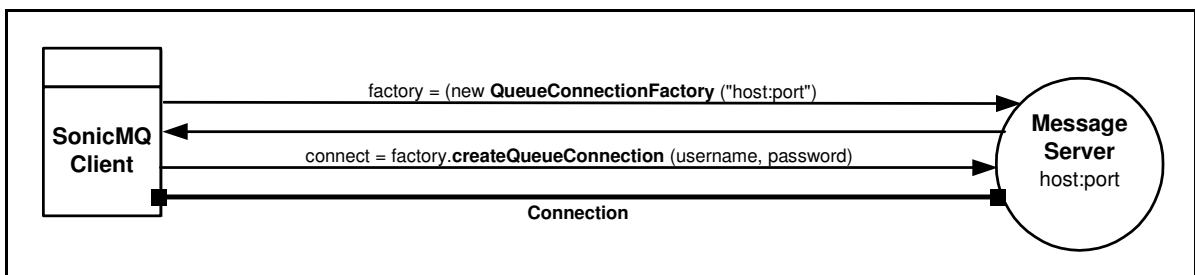


Figure 19. Using a Constructor to Create a ConnectionFactory Object

The syntax for instantiating the object class in each of the two messaging models is:

```
PTP:    javax.jms.QueueConnectionFactory factory =
          (new progress.message.jclient.QueueConnectionFactory
           (parameters);
```

```
Pub/Sub: javax.jms.TopicConnectionFactory factory =
            (new progress.message.jclient.TopicConnectionFactory
             (parameters);
```

There are several supported constructors for creating a `ConnectionFactory` object in each messaging domain. The constructors use combinations of the **URL**, **connectID**, **username** and **password** parameters.

The parameters of a constructor for an instance of a `ConnectionFactory` object in both PTP and Pub/Sub domains are identical:

- **URL** – The Uniform Resource Locator of the message server where the connection is intended (in the form `[protocol : //]hostname[: port]`). where:
 - `protocol` is the message server’s communication protocol (`[HTTP | SSL | TCP]`).
 - `hostname` is a networked SonicMQ server machine (or **local host** if the client and server are on the same machine).
 - `port` is the port on the host where the message server is listening. The message server’s default port value is **2506**.
- **connectID** — The `ConnectID` is a SonicMQ identifier that can control whether the message server allows multiple connections under a user-name and `ConnectID` combination:
 - If the `ConnectID` is not `null`, only one connection with that particular user-name and `ConnectID` can be established.
 - If the `ConnectID` is `null`, any number of connections can be established with the same username.

You might create a `ConnectID` by combining the username with some additional identifier.

- **username** and **password** — The user authentication that is enforced when the security database is active. These parameters are optional. When both parameters are omitted, they both default to `""`, an empty string.

Under the SSL protocol, client authentication can be achieved by retrieving the username from the client certificate. In that case you simply pass the special-purpose username **AUTHENTICATED**. The password is ignored.

Note When user identification is omitted when creating a connection, the connection uses the default values from the `ConnectionFactory`. If true authentication with the security database is implemented and the user name is invalid, a `javax.jms.JMSSecurityException` is thrown.

You can use the common name from a certificate when you use SSL mutual authentication. See the *SonicMQ Deployment Guide* for more about SSL and security.

Load Balancing and Failover Lists

Your client applications can create failover and load-balanced connections to message servers using options set in `ConnectionFactory` objects. See JavaDoc for the factories for more information.

► To implement failover:

1. Create a comma-separated list of server URLs. The client will attempt to connect to servers in this list.
2. Add a statement where `factory.setConnectionURLs(serverList)` points to the text list you created. The client will connect to the first available message server on the list, overriding the `URL` parameter in the `QueueConnectionFactory` or `TopicConnectionFactory` constructor which specifies a single message server.
3. Add a statement where `factory.setSequential(boolean)` sets whether to start with the first name in the list (`true`) or a random element (`false`).

► To implement load balancing:

- Add a statement where `factory.setLoadBalancing` is set to `true` in your application.

With client-side load balancing enabled, a connect request can re-directed to another message server within a SonicMQ cluster, provided load-balancing has not been disabled on the server side.

► **To check failover settings:**

1. In an application, add a statement with `factory.getConnectionURLs()` to return the server list.
2. Add a statement with `factory.getSequential()` to return the boolean indicator of whether the list is used sequentially or randomly.

► **To check load balancing settings:**

1. In an application, add a statement with `factory.getLoadBalancing()` to return the boolean indicator of whether load-balancing is enabled by the client.
2. Get the URL of the server that the client connects to as a result of load balancing by calling `getBrokerURL()` on the connection object, not the factory object.

Connection

After instantiating a `ConnectionFactory` object, the factories' `create` methods are used to create a connection. The first action a client must take is to identify and establish connection with a message server. The following constructors use a connection factory object to get the connection.

Important An application should not use a Java constructor to create connections directly.

createQueueConnection

A `QueueConnection` is an active PTP connection to a SonicMQ message server. A client application will use the `QueueConnection` to create one or more `QueueSessions`, the threads for sending messages to a specified queue and receiving messages from specified queues.

There are two variants of the create method:

- Use the default user name and password:

```
connect = factory.createQueueConnecti on ()
```

- Supply the preferred user name and its authenticating password:

```
connect = factory.createQueueConnecti on
                (String user-name, String password)
```

createTopicConnection

A TopicConnection is an active Pub/Sub connection to a SonicMQ message server. A client application will use the TopicConnection to create one or more TopicSessions, the threads for publishing messages and receiving messages from subscribed topics. There are two variants of the create method:

- Use the default user name and password:

```
connect = factory.createTopicConnecti on ()
```

- Supply the preferred user name and its authenticating password:

```
connect = factory.createTopicConnecti on
                (String user-name, String password)
```

Connection Retry

You can write a connection retry process that lets the thread sleep for a specified time before retrying the connection. The following example is a retry of a queue connection:

```
try
{
    System.out.println("Attempting to create connection...");
    connect = factory.createQueueConnecti on (username, password);
} catch (j avax. j ms. JMSEExcepti on j mse)
{
    System.out.print("Cannot connect to server: " + server + ". ");
    System.out.println
        ("Pausing " + CONNECTION_RETRY_PERIOD / 1000 + "
         seconds before retry.");
        try
        {
            Thread.sleep(CONNECTION_RETRY_PERIOD);
        }
        catch (j ava. l ang. InterruptedExcepti on i e) { }
    conti nue;
}
}
```

Session

A JMS Session represents a single thread of activity. All actual messaging is done through the session object. Each message handler is associated with a single session (there could be multiple message handlers in a session, or none at all). A session is associated with the `Connection` object.

The session interface available is determined by the messaging paradigm chosen for the connection. The syntax for creating a session on an established connection is:

- **Point-to-Point**

```
javax.jms.QueueSession createQueueSession  
(boolean transacted, int acknowledgeMode)
```

- **Publish and Subscribe**

```
javax.jms.TopicSession createTopicSession  
(boolean transacted, int acknowledgeMode)
```

where:

- *transacted* — [`true` | `false`] if `true`, the session will be transacted.
- *acknowledgeMode* — [`AUTO_ACKNOWLEDGE` | `CLIENT_ACKNOWLEDGE` | `SINGLE_MESSAGE_ACKNOWLEDGE` | `DUPS_OK_ACKNOWLEDGE`]

Indicates whether the consumer or the client will acknowledge any messages it receives.

The parameters of the session are qualified so that when a session is transacted, the `acknowledgeMode` is ignored (even though required). Similarly, `acknowledgeMode` has no effect when a session is only producing messages.

Explicit Acknowledgement

While all JMS messages support using the `acknowledge()` method, only some session modes allow a message to be explicitly acknowledged.

The effect of explicit acknowledgement is:

- When the session acknowledgement mode is `CLIENT_ACKNOWLEDGE`, all messages previously received by the session are acknowledged.

- When the session acknowledgement mode is `SINGLE_MESSAGE_ACKNOWLEDGE`, only the current message is acknowledged.
- When the session acknowledgement mode is `AUTO_ACKNOWLEDGE`, calls to `acknowledge()` are ignored.
- When the session acknowledgement mode is `DUPS_OK_ACKNOWLEDGE`, calls to `acknowledge()` are ignored.
- When the session is transacted, calls to `acknowledge()` are ignored.

Acknowledgement Mode

Communication between the message server and the message consumer involves an indication of receipt of the message. One of the following acknowledgement modes is enforced for all messages in a session:

- `AUTO_ACKNOWLEDGE` — The session automatically acknowledges the client's receipt of a message by successfully returning from a call to `receive()` (**synchronous** mode) or when the session `MessageListener` successfully returns (**asynchronous** mode). The last message might be redelivered.
- `CLIENT_ACKNOWLEDGE` — An explicit `acknowledge()` on a message acknowledges the receipt of all messages that have been produced and consumed by the session that gives the acknowledgement. When a session is forced to recover, it restarts with its first unacknowledged message.
- `SINGLE_MESSAGE_ACKNOWLEDGE` — An explicit `acknowledge()` on a message acknowledges only the current message and no preceding messages. This mode is a SonicMQ extension to the JMS standard.
- `DUPS_OK_ACKNOWLEDGE` — The session “lazily” acknowledges the delivery of messages to consumers, possibly allowing some duplicate messages after a system outage.

Warning While acknowledgement sets standards for delivery from the client to the message server, there is no reply to the sending application. If an application requires a reply to the sender, use the `JMSReplyTo` header field to indicate the request and program your application to respond to this header field. The requestor can also append a correlation identifier that will ensure that the reply matches its request.

Recover

A client might build up a large number of unacknowledged messages while attempting to process them. A session's `recover()` method is used to stop a session and restart it with its first unacknowledged message.

A `recover()` action event tells SonicMQ to stop message delivery in the session, set the `redelivered` flag on unacknowledged messages it will redeliver under the recovery, and then resume (“playback”) delivery of messages, possibly in a different order than originally delivered.

The need for the `recover()` method is most apparent when the acknowledgement mode is `CLIENT_ACKNOWLEDGE` or `SINGLE_MESSAGE_ACKNOWLEDGE`.

Transacted Sessions

When a session is **transacted**, that session will combine a group of one or more messages with client-to-message server **ACID** properties:

Atomic, Consistent, Isolated, and Durable.

When a session is transacted, message input and message output are staged on the message server system but not completed until you call the method to complete the transaction. Completion of a transaction, determined by your code, does one of the following:

- **Commit** — The series of messages is sent to consumers.
- **Roll Back** — The series of messages (if any) is destroyed.

The completion of a session's current transaction automatically begins the next transaction. Transacted sessions impact producers and consumers in the ways described in [Table 5](#).

Table 5. Transacted Session Events by Message Role

<i>Role</i>	<code>commit()</code>	<code>rollback()</code>
Producer	Send the series of messages staged since the last call.	Dispose of the series of produced messages staged since the last call.
Consumer	Dispose of the series of messages received since the last call.	Redeliver the series of received messages retained since the last call.

When a rollback is done in a session that is both sending and receiving, its produced messages are destroyed and its consumed messages are automatically recovered.

To check whether a session is transacted use the `getTransacted()` method. The return value is `true` if the session is in transacted mode.

Session Objects

The primary session objects allow access to the destinations, producers, consumers, and messages that are used in the session, as shown in [Figure 20](#).

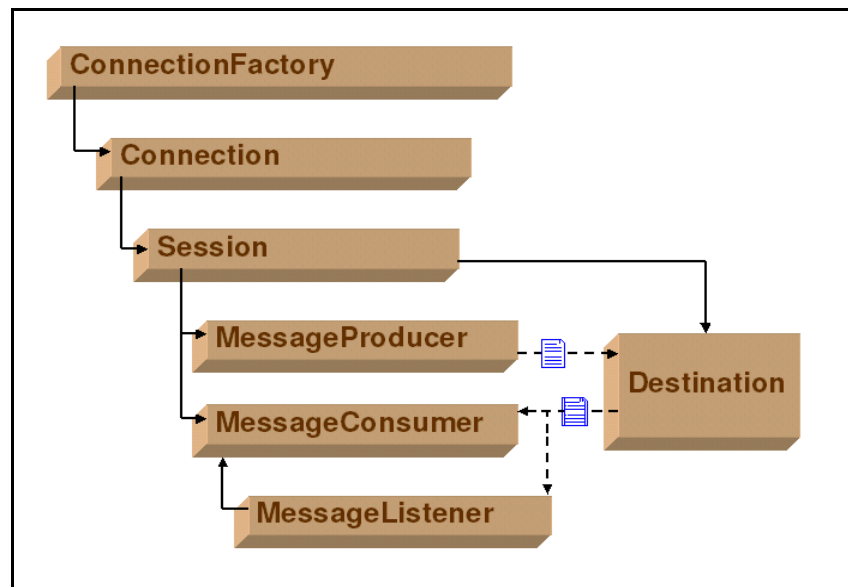


Figure 20. Primary Session Objects

[Figure 21](#) shows the types of message objects that are created from session methods. The message types are common and extended into both JMS

domains. Note that the `XMLMessage` type is unique to SonicMQ and is an extension of the `TextMessage` type.

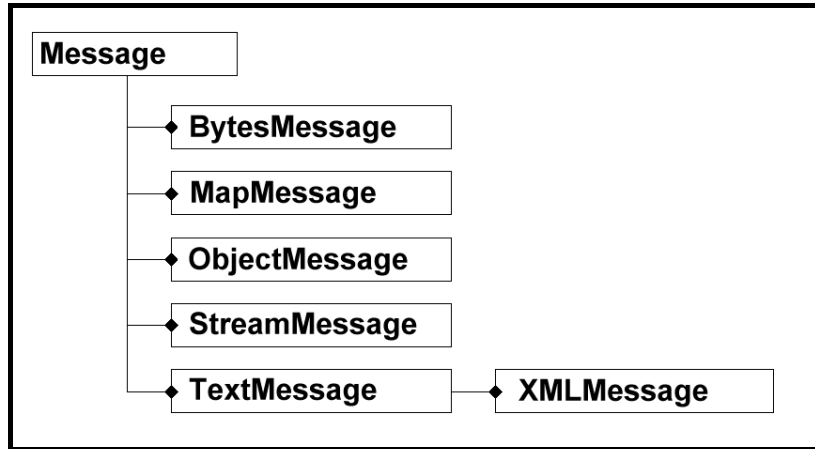


Figure 21. Types of SonicMQ Message Objects

create [Destination]

Destinations are administered objects that can be controlled by an administrator and can be retrieved through JNDI or other object storage mechanisms.

See “*JMS Administered Object Stores*” in the *SonicMQ Installation and Administration Guide* to learn how the SonicMQ Explorer allows you to create destinations in both JNDI and file stores.

The destination object created can be a queue or a topic.

Point-to-Point: createQueue

```
javax.jms.QueueSession queue = session.createQueue(queueName)
```

where:

- `queueName` — A String name that has been established in the message server message store. If security is defined for queues, the user might be constrained from reading or writing to a queue.

Publish and Subscribe: createTopic

```
javax.jms.TopicSession topic = session.createTopic(topicName)
```

where:

- *topicName* — An arbitrary String name for the topic consisting of at most 256 Unicode characters that does not contain the reserved characters period (.), pound (#), dollar sign (\$), asterisk (*), or double colon (: :), and does not begin with the string “SonicMQ.”. If security is defined for topics, the user might be constrained from reading or writing at a topic content node.

Using a Lookup for Destinations

While topics and queues are administered objects, there are advantages to programmatic lookup of defined destinations.

SonicMQ lets you store administered objects in some object store—JNDI or a simple file store—and then reference the object indirectly (by name) in some context.

See [Chapter 12, “Lookup of Administered Objects,”](#) for more information.

Temporary Destinations

Temporary destinations (`TemporaryTopic` or `TemporaryQueue`) can be created for request-and-reply mechanisms.

See [“Reply-to Mechanisms” on page 153](#) for more information.

create [MessageProducer]

The Producer interface is created from a Session method.

Point-to-Point: createSender

```
sender = sendSession.createSender(queue);
```

Publish and Subscribe: createPublisher

```
publisher = pubSession.createPublisher(topic);
```

create [MessageConsumer]

The Consumer interface is created from a Session method in the form:

Point-to-Point: createReceiver

```
javax.jms.QueueReceiver qreceiver =  
    recvSession.createReceiver(queue);
```

Publish and Subscribe: create[Durable]Subscriber

- Regular subscription:

```
javax.jms.TopicSubscriber subscriber =  
    subSession.createSubscriber(topic);
```

- Durable subscription (See [“Durable Subscriber” on page 196.](#)):

```
javax.jms.TopicSubscriber subscriber =  
    subSession.createDurableSubscriber(topic, subscriber);
```

create [Message]

The message type is created from a Session method in the general form:

```
javax.jms.[type]Message msg = sendSession.create[type]Message()
```

where *type* is the JMS message type. Specifically:

- `javax.jms.TextMessage msg = sendSession.createTextMessage()`
- `javax.jms.BytesMessage msg = sendSession.createBytesMessage()`
- `javax.jms.MapMessage msg = sendSession.createMapMessage()`
- `javax.jms.Message msg = sendSession.createMessage()`
- `javax.jms.ObjectMessage msg = sendSession.createObjectMessage()`
- `javax.jms.StreamMessage msg = sendSession.createStreamMessage()`

The `XMLMessage` type, as a SonicMQ extension to the JMS standard, is not created from a `javax.jms.session`—the session must be cast to a `progress.message.jclient.session` first, as in:

- `progress.message.jclient.XMLMessage xMsg =
 sendSession.createXMLMessage()`

Message interfaces, structure, and fields are detailed in [Chapter 4, “Messages.”](#)

Starting, Stopping, and Closing Connections

Connections require an explicit `start` command to begin the delivery of messages. All sessions within a connection respond concurrently to the Connection `start`, `stop`, and `close` events.

While many session and connection objects have individual `close` methods, these are usually viewed as ways to recapture resources and were not previously discussed in terms of how they interact when the Connection's state changes.

connect.start

Start delivery of incoming messages through a connection. Under a restart, delivery begins with the oldest unacknowledged message. Starting an already started session is ignored:

```
connect.start()
```

connect.stop

Stop delivery of incoming messages through a connection. After stopping, no messages are delivered to any message consumers under that connection. If synchronous receivers are used, they will block. A stopped connection can still send or publish messages. Stopping an already stopped session is ignored:

```
connect.stop()
```

Behavior of Producers and Consumers in a Stopped Connection

When a connection is stopped, it is in effect paused. The message producers continue to perform their function. The consumers, however, are not active until the connection restarts. When the stop method is called, the stop will wait until all the message listeners have returned before it returns. Receivers that are active still have their timers running and can receive null messages.

connect.close

When a connection is closed, all message processing within the connection's one or more sessions is terminated. If there was a message available at the time of the close, the message (or a null) can be returned, but the message consumer might get exceptions by trying to use facilities within the closed connection. Closing a closed connection has no effect and does not throw an exception:

```
connect.close()
```

Behavior of Producers and Consumers in a Closed Connection

When a connection is closed, all message processing within the connection's one or more sessions is terminated. If there was a message available at the time of the close, the message (or a null) can be returned, but the message consumer might get exceptions by trying to use facilities within the closed connection.

When a transacted session is closed, the transaction in progress is marked as a rollback.

The message objects can be used in a closed connection with the exception of the message's acknowledge methods.

Closing a Session

While each connection can have many sessions, each session is a single thread of execution. When the connection starts, stops, or closes all its sessions are impacted.

The `close` method is the only `Session` method that can call for a different thread in the connection to close while some other session method is being executed in another thread.

Closing a `CLIENT_ACKNOWLEDGE` session does not force an `acknowledge()` to occur. Attempts to use a closed connection's sessions or session objects throws an `IllegalStateException`. Starting a started connection or closing a closed connection has no effect and does not throw an exception.

Note The message objects can be used in a closed connection with the exception of the message's acknowledge methods.

Flow Control

The asynchronous benefits of SonicMQ are not limited to simply receiving without blocking. They also include:

- Send and receive buffers that stage messages in transit between a client application and a message server
- An optimized persistence mechanism to maximize server performance for guaranteed message delivery
- **Concurrent Transacted Cache** technology that uses in-memory cache and high-speed log files to increase throughput for short-duration persistent messages
- Queues defined with specified amounts of memory and disk space reserved for the queue content

Any of these resources might be offered more data than can be managed. If flow control is active, SonicMQ will throttle back the message flow from the producer, allowing the next message to flow into the buffers only when space is available.

- In Pub/Sub you can disable flow control so that when resources are nearly exhausted, SonicMQ can, under programmatic control, throw exceptions until flow control conditions are cleared.
- In PTP flow control is always active.

When flow control is active in Pub/Sub, the messages might be sent to subscribers at a rate that is faster than that at which the messages are actually consumed. When the buffers that store unprocessed messages approach capacity, flow control can stop new additions until the buffers fall below a stated level.

The back pressure from slower consumption starts to impact the buffers for queues or durable subscriptions. When system or queue capacities are filled with messages in process, flow control is activated against producers. The message acceptance rate drops which eventually results in back pressure at the producers, causing them to either tolerate the slowdowns or, in Pub/Sub with flow control disabled, to throw an exception so that you can handle the

situation, for example by catching the exception and having the application wait some period of time before re-publishing.

To avoid the invocation of flow control you can:

- Optimize application processing on incoming messages.
- Adjust the consumer buffer.
- Increase the size of queues.
- Decrease the message expiration time of messages.

Note

Messages sent to a queue will only expire after they have been placed on the queue, so expiration detection can only result from:

- Dequeue operations by receivers
- Processing by the queue cleanup thread
- Browsing the queue

In the Pub/Sub domain you can disable flow control so that your application can catch exceptions to determine when messages are being published at too fast a rate by setting the following property in `TopicSession`:

```
setFlowControlDisabled(boolean disabled)
```

where **TRUE** indicates that flow control is not active.

Using Multiple Connections, Sessions, and Consumers

There are many advantages to using multiple connections and multiple sessions in an application even though the ordering of messages is only assured within a session—a single thread of execution.

Multiple Connections

Multiple connections work concurrently. If your application requires concurrency, you should first consider using one connection and multiple sessions because connections require more resources compared to sessions. The most common reason for having two connections in one client application is to support both queues (PTP) and topics (Pub/Sub).

Sometimes the sheer volume of information flowing through the connection warrants multiple connections rather than multiple sessions. [Figure 22](#) shows two connections, each with one session.

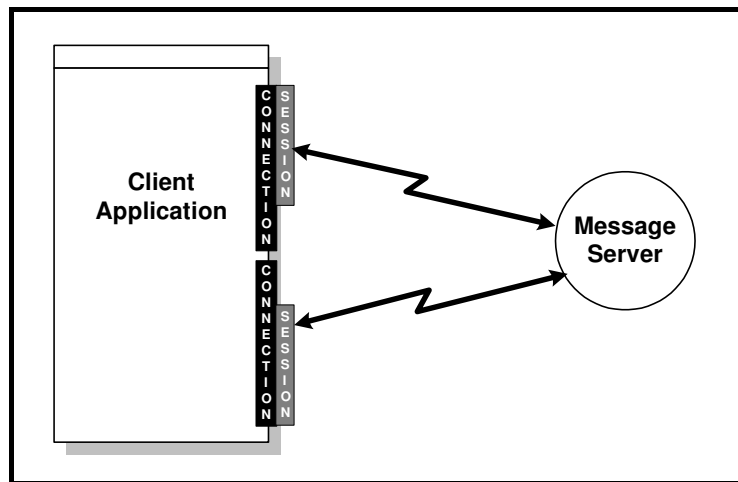


Figure 22. Multiple Connections in a Client Application

Multiple Sessions on a Connection

Using multiple sessions gives up the benefits of serialized operations on a single thread of execution. Multiple sessions are best suited for alternate or supporting functions within an application. [Figure 23](#) shows multiple sessions using two sessions and only one connection. As the connection is associated with a messaging domain—PTP or Pub/Sub—multiple sessions are constrained to the connection’s domain.

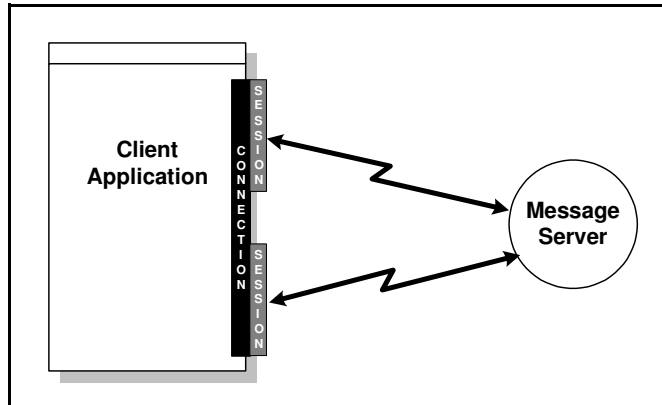


Figure 23. Multiple Sessions on a Connection

Coding Connections and Sessions

Each of the SonicMQ sample applications uses client connections and sessions. The following code sections are excerpts from the Pub/Sub Durabl eChat sample:

```
public class Durabl eChat implements
    javax.jms.MessageListener, // to handle message subscriptions
```

They demonstrate how to:

- Get a connection and session.
- Create session objects.
- Set up listeners and start the connection.
- Handle exceptions on the connection.

Get a Connection and Session

This ReliableChat `setupConnection` snippet uses the `ConnectionFactory` and uses active pings to check the pulse of the connection:

```
// Get a connection factory
javax.jms.TopicConnectionFactory factory = null;
try
{
    factory = (new progress.message.jclient.TopicConnectionFactory (m_broker));
} catch (javax.jms.JMSException jmse) ...
// Wait for a connection.
while (connect == null)
{
    try
    {
        System.out.println("Attempting to create connection...");
        connect = (progress.message.jclient.TopicConnection)
            factory.createTopicConnection (m_username, m_password);
        ...
        // Ping the broker to see if the connection is still active.
        connect.setPingInterval (30);
    } catch (javax.jms.JMSException jmse)
    {
        System.out.print("Cannot connect to broker: " + m_broker );
        System.out.println
            ("Pausing " + CONNECTION_RETRY_PERIOD / 1000 + "
             seconds before retry.");
    }
    try
    {
        Thread.sleep(CONNECTION_RETRY_PERIOD);
    } catch (java.lang.InterruptedException ie) { }
    continue;
}
...
```

Using Active Pings to Monitor the Health of the Connection

In the preceding code segment the statement `connect.setPingInterval(6)` indicates the use of a method that lets the application detect when a connection gets dropped by setting a `PingInterval` of six seconds. The **active pings** are a SonicMQ feature that allows an application to check the presence and alertness of the message server on a connection. This technique is particularly useful for connections that listen for messages, but do not send messages.

This feature is limited to interruption in the physical connection. If the message server stops on a good connection—with or without pings—an exception is generated.

Invoking `setPingInterval(interval_in_seconds)` on a connection sends a ping message to the message server on that connection at the specified interval to examine the health of the connection.

Note Avoid setting a small ping interval. This wastes cycles and your application will be burdened with temporary network unavailability.

Create Session Objects and the Listeners

Two sessions are created in this continuation of the `ReliableChat` sample, one session to work with the standard input and send functions and the other to work with the message listener and the messages it delivers for consumption. Each session declares its acknowledgement mode then sets up the destination and the publisher or subscriber. The message listener is activated against the consumer destination. The detailed code is as follows:

```
// Create the Sessions, Publisher, Subscriber, and Topics
try
{
...
pubSession = connect.createTopicSession
                (false, javax.jms.Session.CLIENT_ACKNOWLEDGE);
subSession = connect.createTopicSession
                (false, javax.jms.Session.CLIENT_ACKNOWLEDGE);
javax.jms.Topic topic = pubSession.createTopic (APP_TOPIC);
```

```

j a v a x . j m s . T o p i c S u b s c r i b e r s u b s c r i b e r =
    s u b S e s s i o n . c r e a t e D u r a b l e S u b s c r i b e r
        ( t o p i c , " S a m p l e S u b s c r i p t i o n " );
s u b s c r i b e r . s e t M e s s a g e L i s t e n e r ( t h i s );
p u b l i s h e r = p u b S e s s i o n . c r e a t e P u b l i s h e r ( t o p i c );
// R e g i s t e r t h i s c l a s s a s t h e e x c e p t i o n l i s t e n e r f o r a n y p r o b l e m s .
c o n n e c t . s e t E x c e p t i o n L i s t e n e r ( ( j a v a x . j m s . E x c e p t i o n L i s t e n e r ) t h i s );
S y s t e m . o u t . p r i n t l n ( " . . . S e t u p c o m p l e t e . " );
. . .

```

Start the Connection

When all the session objects and settings are established, the ReliableChat connection is started.

```
connect.start();
```

Messages are composed and sent by the publisher session. Messages are delivered and consumed by the subscriber session.

Handle Exceptions on the Connection

The exception handler can handle errors actively as in this ReliableChat snippet where a connection problem initiates a reconnection routine:

```

// H a n d l e a s y n c h r o n o u s p r o b l e m w i t h t h e c o n n e c t i o n .
p u b l i c v o i d o n E x c e p t i o n ( j a v a x . j m s . J M S E x c e p t i o n j s m e )
{
    // S e e i f c o n n e c t i o n w a s d r o p p e d .
    // T e l l t h e u s e r t h a t t h e r e i s a p r o b l e m .
    S y s t e m . e r r . p r i n t l n ( "\n\nT h e r e i s a p r o b l e m w i t h t h e c o n n e c t i o n . " );
    S y s t e m . e r r . p r i n t l n ( "    J M S E x c e p t i o n : " + j s m e . g e t M e s s a g e ( ) );
    // I f t h e e r r o r i s a d r o p p e d c o n n e c t i o n , t r y t o r e c o n n e c t .
    // N O T E : t h e t e s t i s a g a i n s t P r o g r e s s S o n i c M Q e r r o r c o d e s .
    i n t d r o p C o d e =
        p r o g r e s s . m e s s a g e . j c l i e n t . E r r o r C o d e s . E R R _ C O N N E C T I O N _ D R O P P E D ;
    i f ( p r o g r e s s . m e s s a g e . j c l i e n t . E r r o r C o d e s . t e s t E x c e p t i o n
        ( j s m e , d r o p C o d e ) )
    {

```

```
System.err.println ("Please wait while the application tries  
to "+ "re-establish the connection...");  
  
// Reestablish the connection  
connect = null;  
setupConnection();
```

Handling Dropped Connection Errors Caught with Active Pings

When message server failure causes a dropped connection, the ensuing `TCP_RESET` fires the `onException()` method of the `ExceptionListener` with the **connection-dropped** error code.

Important Ping also can also detect the drop of connection due to network failure, such as a disconnected cable.

When a message server is disconnected from the network, an exception is thrown to keep a publisher from locking up. However, a subscriber will not detect such an error since no `TCP_RESET` is sent unless the active ping feature has been enabled on the connection.

Exception Listeners are Not Intended for JMS Errors

The `ExceptionListener` is a way to pass information about a problem with a connection by calling the listener's `onException()` method, passing it a `JMSException` describing the problem.

This allows a client to be asynchronously notified of a problem. Some connections only consume messages so they would have no other way to learn that their connection has failed.

The exceptions delivered to `ExceptionListener` are those that do not have any other place to be reported. If an exception is thrown on a JMS call it, by definition, must not be delivered to an `ExceptionListener` — in other words, the `ExceptionListener` is not for the purpose of monitoring all exceptions thrown by a connection.

JMS Messaging Domains

The JMS messaging domains are primarily differentiated by messaging behaviors. The functionally viewed by the programmer is quite similar, as shown in their interfaces and methods in [Table 6](#).

Table 6. Connected Session Functionality Common to PTP and Pub/Sub

<i>javax.jms Interface</i>	<i>Functionality in Either Domain</i>
ConnectionFactory extended by: QueueConnecti onFactory Topi cConnecti onFactory	<ul style="list-style-type: none"> ■ Allows administrative control of communication resources ■ Creates one or more Connecti ons
Connection extended by: QueueConnecti on Topi cConnecti on	<ul style="list-style-type: none"> ■ Creates one or more Sessi ons ■ Supports concurrent use ■ Lets applications specify name-password for client authentication ■ Allows unique client identifiers ■ Provides Connecti onMetaData ■ Supports an Excepti onLi stener ■ Provides Start and Stop methods ■ Provides a cl ose method for connections
Session extended by: QueueSessi on Topi cSessi on	<ul style="list-style-type: none"> ■ Serves as a factory for MessageProducers and MessageConsumers ■ Sessi ons and Desti nati ons are used to create multiple MessageProducers and MessageConsumers ■ Serves as a factory for TemporaryDesti nati ons ■ Creates Desti nati on objects with dynamic names ■ Serves as a factory for Messages ■ Supports serial order of messages consumed and produced ■ Retains consumed messages until acknowledged ■ Serializes execution of registered MessageLi steners ■ Provides a cl ose method for sessions

About Messages

The message is the essence of the SonicMQ Messaging Server. A message is a package of bytes that encapsulates the message body as a payload and then exposes metadata that identify—at a minimum—the message, its timestamp, its priority, its destination, and the type of message enclosed.

When a text message is published, it might be coded as:

```
private void jmsPublish (String aMessage)
    javax.jms.TextMessage msg = session.createTextMessage();
    msg.setText( user + ": " + aMessage );
    publisher.publish( msg );
```

When a message is received it might be through an asynchronous listener:

```
// Handle an asynchronously received message
public void onMessage( javax.jms.Message aMessage)
    { ...
    // Cast the message as a text message.
    javax.jms.TextMessage textMessage = (javax.jms.TextMessage) aMessage;
    // Read a single String from the text message, print to stdout.
    String string = textMessage.getText();
    ...
    }
```

Message Type

The JMS specification defines five types of messages, all derived from the `Message` interface, which also defines message headers and the `acknowledge` method used by all JMS messages. SonicMQ provides an XML message type as an extension of the JMS Text type. Figure 24 diagrams the SonicMQ message types.

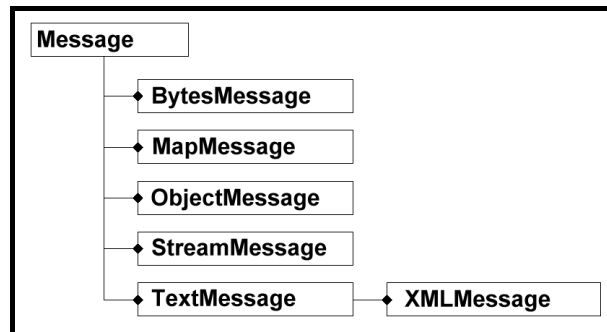


Figure 24. SonicMQ Message Types

The message types can be described as follows:

- **Message** — The type `Message` is the root interface of all JMS messages. It contains no body, but does hold all the standard message header information. It can be sent when a message containing only header information is sufficient.
- **BytesMessage** — A stream of uninterpreted bytes. This message type exists to support cases where the contents of the message will be shared with applications that cannot read Java types or 16-bit Unicode encodings. It is also useful when the information to send already exists in binary form.
- **MapMessage** — A set of name-value pairs where names are strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name. An example of `MapMessage` usage is a message describing a new product, which includes the price, weight, and description; the names in the `MapMessage` correlate to columns in a database table in which the consumer stores the information.

- **ObjectMessage** — A message that contains a serializable Java object. An ObjectMessage is useful when both JMS clients are Java applications or applets with access to the same class definition.
- **StreamMessage** — A stream of Java unkeyed primitive values that is filled and accessed sequentially. Since a StreamMessage contains only raw data and no keys, it takes up less space than an equivalent MapMessage.
- **TextMessage** — A message containing a java.lang.String or String. Use a TextMessage when the message content does not require any particular structure, for example when the message body is simply printed or copied by the consumer.
- **XMLMessage** — A message containing a string representing the XML tree that can be parsed as an XML document.

Creating a Message

The message *type* is created from a Session method in the form:

```
j avx. j ms. [ type] Message msg = sessi on. create[ type] Message ()
```

such that the set of methods is:

- j avx. j ms. Message msg = sessi on. createMessage ()
- j avx. j ms. BytesMessage msg = sessi on. createBytesMessage ()
- j avx. j ms. MapMessage msg = sessi on. createMapMessage ()
- j avx. j ms. ObjectMessage msg = sessi on. createObjectMessage ()
- j avx. j ms. StreamMessage msg = sessi on. createStreamMessage ()
- j avx. j ms. StreamMessage msg = sessi on. createStreamMessage (Serializable object)
- j avx. j ms. TextMessage msg = sessi on. createTextMessage ()

The XMLMessage type, described below, extends the TextMessage type.

XML Type

There is a slight difference for the XMLMessage type extension that is unique to SonicMQ:

```
progress. message. j cli ent. XMLMessage xMsg =
    ((progress. message. j cli ent. Sessi on) sessi on).
    createXMLMessage ()
```

An `XMLMessage` is an extension of the `TextMessage` and is used to send a message containing XML text in a `java.lang.String`.

`TextMessage` inherits from `Message`, adding a text message body. `XMLMessage` then allows access to the XML text's Document Object Model (DOM).

An implementation of the DOM interface is instantiated when the developer wants to access the XML message body using the Document Object Model (DOM), a standard (WC3) application programming interface for accessing, updating, and creating XML documents.

Message Structure

JMS Messages are composed of the following parts:

- **Header Fields (JMS)** — All messages support the same set of header fields. Header fields contain values used by clients and message servers to identify and route messages.
- **User-defined Properties** — User-defined name-value pairs that can be used for filtering and application requirements.
- **Provider-defined Properties** — Properties defined and typed by SonicMQ for carrying information used by SonicMQ features.
- **Supported JMS-defined Properties (JMSX)** — Predefined name-value pairs that are an efficient mechanism for supporting message filtering.
- **Body** — JMS defines several types of message body, which cover the majority of messaging styles currently in use.

Messages and Selectors

The JMS message system provides programmatic access to all components of a message. However, any content that should be exposed to the subscriber for message selection or routing must be enclosed in the appropriate header fields and properties, as the message body cannot be accessed for selection and routing data.

Message Header Fields

The message header fields are defined and used by the sender and the message server to convey basic routing and delivery information.

The message header fields are described in detail in [Table 7](#).

Table 7. Message Header Fields

<i>JMS Header field</i>	<i>Type</i>	<i>Description</i>	<i>Usage</i>	<i>Comments</i>
<p>JMSDestination</p> <p>Required.</p> <p>Set by the producer send/publish method.</p>	String	The destination where the message is being sent.	<p>While a message is being sent this value is ignored.</p> <p>After completion of the publish send method, it holds the destination specified by the send.</p>	When a message is received, its destination value must be equivalent to the value assigned when it was sent.
<p>JMSDeliveryMode</p> <p>Required.</p> <p>Set in a producer send/publish parameter.</p>	String	Specifies whether the message is to be retained in the message server's database.	<p>Required.</p> <p>Must be PERSISTENT, NON_PERSISTENT, or NON_PERSISTENT_ASYNC</p>	Default value is (NON_PERSISTENT) .
<p>JMSMessageID</p> <p>Required.</p> <p>Set by the producer send/publish method.</p>	String	SonicMQ field for a unique identifier.	A message ID value must start with "ID: ".	While required, the algorithm that calculates the ID on the client can be bypassed which sets the JMSMessageID to null .
<p>JMSTimestamp</p> <p>Required.</p> <p>Set by the producer send/publish method.</p>	Long	GMT time on the producer system clock when the message was sent.		Set method exists but is always overridden by the send method valuation.

Table 7. Message Header Fields (continued)

JMS Header field	Type	Description	Usage	Comments
JMSCorrelationID Optional. Set by producer mutator method.	String	Message server-specified message ID or an application-specific String	Required when other JMS providers support the native concept of a correlation ID.	An application made up of several clients might want an application-specific value for linking messages.
JMSCorrelationID AsBytes Optional. Set by producer mutator method.	bytes	A native byte[] value.		
JMSReplyTo Optional. Set by producer mutator method.	String	The destination where a reply to the current message should be sent.	If null, no reply is expected. If not null, expects a response, but the actual response is optional and the mechanism must be coded by the developer.	Message replies often use the CorrelationID to assure that replies synchronize with the requests.
JMSRedelivered Set by message server.	boolean	If true it is likely that this message was delivered to the client earlier but the client did not acknowledge its receipt at that time.	Set at the time the message is delivered.	When acknowledgement is expected and not received in a specified time, the message server can decide to set this and resend.
JMSType Optional. Set by producer mutator method.	String	Contains the name of a message's definition as found in an external message type repository.	Recommended for systems where the repository needs the message type sent to the application.	This is not, by default, the message type.

Table 7. Message Header Fields (continued)

JMS Header field	Type	Description	Usage	Comments
<p>JMSExpiration</p> <p>Required.</p> <p>Set by the producer send/publish method by incrementing the current GMT time on the producer system by the producer send/publish parameter, <code>timeToLive</code>.</p>	long	When a message's expiration time is reached, the message server can discard it. Clients should not receive messages that have expired; however, the JMS specification does not guarantee that this will not happen.	<p>The sum of the time-to-live value specified by the client and the GMT at the time of the send. If the time-to-live is specified as zero, the message does not expire.</p> <p>Default value is 0.</p>	<p>When a message is sent, expiration is left unassigned. After completion of the send method, it holds the expiration time of the message.</p> <p>Default value is 0.</p> <p>The expiration of a message can be managed by setting the message property <code>JSM_SonicMQ_preserveUndelivered</code> which will transfer an expired (or undeliverable) message to the message server's Dead Message Queue.</p>
<p>JMSPriority</p> <p>Required.</p> <p>Set in a producer send/publish parameter.</p>	int	Sets a value that will allow a message to move ahead of other undelivered messages in a topic or queue. Also allows message selectors to pick messages at a given priority.	<p>A ten-level priority value with 0 as the lowest priority and 9 as the highest.</p> <p>0 to 4 are normal.</p> <p>5 to 9 are expedited.</p> <p>Default value is 4.</p>	The JMS specification does not require that SonicMQ strictly implement priority ordering of messages; however, the message server will do its best to deliver expedited messages ahead of normal messages.

Setting Header Values When Sending/Publishing

The basic method for producing a message allows essential delivery information to accept the JMS default values, for example:

```
publisher.publish(Message message)
```

Default Header Values

Three of the message header fields have default values as static final variables:

- `DEFAULT_DELIVERY_MODE = NON_PERSISTENT`
- `DEFAULT_PRIORITY = 4`
- `DEFAULT_TIME_TO_LIVE = 0`

The default header field values can be changed in the signature of the send or publish method to override the defaults:

- **Point-to-Point:**

```
sender.send(Message message,  
            int deliveryMode,  
            int priority,  
            long timeToLive)
```

- **Publish and Subscribe:**

```
publisher.publish(Message message,  
                  int deliveryMode,  
                  int priority,  
                  long timeToLive)
```

If you use this format of the method but do not intend to override some of the default values, you can substitute them back into the parameter list. For example:

```
private static final int MESSAGE_LIFESPAN = 1800000;  
// milliseconds (30 minutes)  
sender.send(msg,  
            javax.jms.DeliveryMode.PERSISTENT,  
            javax.jms.Message.DEFAULT_PRIORITY,  
            MESSAGE_LIFESPAN);
```

Message Properties

Properties are optional fields that are associated with a message. No message properties are required for any message producer. The property values are used for message selection criteria and data required by applications and other messaging infrastructures. The order of property values is not defined.

Although the JMS specification does not define a policy for what should or should not be made a property, application developers should note that data is handled in a message's body more efficiently than data in a message's properties. For best performance, applications should only use message properties when they need to customize a message's header. The primary reason for doing this is to support customized message selection.

User-defined Properties

A message contains a built-in facility for supporting application-defined property values. In effect, this provides a mechanism for adding application-specific header fields to a message.

Property names must obey the rules for a message-selector identifier. Property values can be `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and `String`.

Property values are set prior to sending a message. When a client receives a message, its properties are in read-only mode. If `clearProperties` is called, the properties are erased and then can be set.

Provider-defined Properties (JMS_SonicMQ)

SonicMQ reserves some property names and declares each property's type. The following properties are prescribed in SonicMQ for use in expressing intended handling of undelivered messages and added data about those messages.

[Table 8](#) lists the SonicMQ defined properties, two set by the message producer to indicate interest in tracking undelivered messages and two set by the message server when a message is transferred into the server’s dead message queue.

Table 8. SonicMQ Provider-defined Properties

<i>JMS Provider-defined Property</i>	<i>Type</i>	<i>Set by</i>
JMS_SonicMQ_preserveUndelivered	boolean	Producer
JMS_SonicMQ_notifyUndelivered	boolean	Producer
JMS_SonicMQ_undeliveredReasonCode	int	Message server
JMS_SonicMQ_undeliveredTimestamp	long	Message server

Review the sample [“Persistent Storage Application \(PTP\)”](#) on page 63 to see how these properties are used. See *SonicMQ Deployment Guide* for detailed information about how these properties contribute to handling undeliverable messages in local message servers and dynamic routing nodes.

JMS-defined Properties (JMSX)

The JMS specification reserves the **JMSX** property name prefix for optional JMS-defined properties. Properties set **on send** are available to the producer and the consumers of the message. Properties set **on receive** are only available to the consumers.

Properties can be referenced in message selectors whether or not they are supported by a connection. They are treated like any other absent property. [Table 9](#) lists and describes the JMSX Message Properties used in SonicMQ. These JMSX properties are set by the producer.

Table 9. JMSX Properties Used in SonicMQ

<i>JMSX Property</i>	<i>Type</i>	<i>Set by</i>
JMSXGroupID	String	Producer on send
JMSXGroupSeq	int	Producer on send

Setting Message Properties

Properties are in no specified order. They might or might not contain values or data extracted from the message body. There are no default properties.

Figure 25 shows the Explorer view of the default JMS header fields and two properties defined by the message sender.

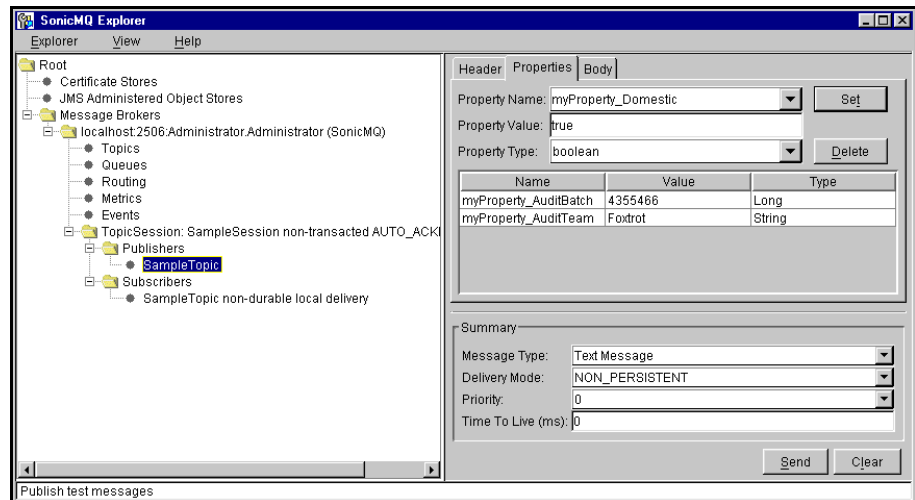


Figure 25. User-defined Properties

Property Methods

JMSX properties can be referenced in message selectors whether or not they are supported by a connection. If values for these properties are not included, they are treated like any other absent property.

Where JMS-defined properties are typed, user-defined properties are typed when they are created by a set method. User-defined properties can be coerced into other data types when they are retrieved.

The setting and getting of message properties allows a full range of data types when the property is established. The properties can be retrieved as a list. A property value can be retrieved by using a get method for the property name.

propertyExists

To check if a property value exists, use the method:

```
public boolean propertyExists(String name)
```

where *name* is the name of the property to test.

Returns `TRUE` if the property exists.

clearProperties

A message's properties are deleted by the `clearProperties` method. This leaves the message with an empty set of properties. Clearing properties effects only those properties that have been defined and has no impact on the header fields or the message body:

```
public void clearProperties()
```

set[type]Property

Message properties are set as name-value pairs where the value is of the declared data type. Setting a property type that does not exist causes it to exist as a property in that message:

```
set[type]Property(String name, [type] value)
```

where *type* is one of the following:

```
[ boolean | byte | short | int | long | float | double | String ]
```

For example, `setBooleanProperty("reconciled", true)`.

getPropertyNames

To iterate through a message's property values, use `getPropertyNames()` to retrieve a property name enumeration. Then use the various property get methods to retrieve their respective values.

get[type]Property

Getting a property value for a property name gets the value of that property. If the property does not exist, a `null` is returned:

```
public [type] get[type]Property(String name);
```

where `type` is one of the following:

```
[ boolean | byte | short | int | long | float | double | String ]
```

For example, `boolean getBooleanProperty("reconciled")` returns `true`.

Property values can be coerced. The accepted conversions are listed in [Table 10](#) where a value written as the row type can be read as the column type.

For example, a `short` property can be read as a `short` or coerced into an `int`, `long` or `String`. An attempt to coerce a `short` into another data type is an error.

Table 10. Permitted Type Conversions for Message Properties

	boolean	byte	short	int	long	float	double	String
boolean	Yes	No	No	No	No	No	No	Yes
byte	No	Yes	Yes	Yes	Yes	No	No	Yes
short	No	No	Yes	Yes	Yes	No	No	Yes
int	No	No	No	Yes	Yes	No	No	Yes
long	No	No	No	No	Yes	No	No	Yes
float	No	No	No	No	No	Yes	Yes	Yes
double	No	No	No	No	No	No	Yes	Yes
String	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Valid coercions are indicated with Yes; those intersections marked with No throw a `JMSException`.

A string-to-primitive conversion might throw a run time exception if the primitives `valueOf()` method does not accept it as a valid string representation of the primitive.

Message Body

The message body has no default value and need not have any content. The message body is populated by the message set method for the message type.

Setting the Message Body

Use the set methods specified by JMS for all types except XML unless the message is read-only (in which case you will need to copy or reset the received message). For example, for a `TextMessage`:

```
msg.setText(aMessage);
```

Setting the Body for an XML Type

The `XMLMessage` type is a message body with XML tags. The `XMLMessage` is a SonicMQ extension to JMS that uses the `setDocument` method to set the body by setting the `org.w3c.dom.Document` object associated with the `XMLDomMessage` contents. This allows the client application to set the contents by passing in a Document Object Model (DOM). The `setDocument` method is written in the form:

```
setDocument(org.w3c.dom.Document aDoc)
```

where `aDoc` is a standard `org.w3c.dom.Document` object.

The `org.w3c.dom.Document` is stored as the internal document for this message. The message content is emptied but will be generated when the message is sent. For best results, the XML Document object should be an instance of the `com.ibm.xml.parser.TXDocument`. However, if it is not a `TXDocument`, it tries to make a node-by-node copy of it.

Important If you use `setText(String string)` where `string` is the string containing the message's data, you set the string containing this message's data, overriding `setText` in class `TextMessage`.

Getting the Message Body

Use the `get` methods required by the JMS specification for all types except XML. For example:

```
msg.getText(aMessage);
```

Getting the Body from an XML Type

For instances of `XMLMessage`, use the `getDocument` method to return an `org.w3c.dom.Document` object created from the `XMLMessage` contents. This will allow the client application to access the contents using the DOM-tree functionality. It is written in the form:

```
org.w3c.dom.Document getDocument()
```

If you use `getText()`, you get the string containing this message's data. If the message has been created with a `setDocument()`, this call will convert it to a text message, overriding `getText` in class `TextMessage`.

About Message Producers and Message Consumers

This chapter describes the generic programming model for messaging that is common to both messaging models, Publish and Subscribe (Pub/Sub) and Point-to-Point (PTP).

Generic Messaging Model

Message producers and message consumers are established in one of the messaging models by creating an appropriate `ConnectionFactory` then creating `Connections`, then `Sessions` on each `Connection` and, finally, the session objects as shown in [Figure 26](#).

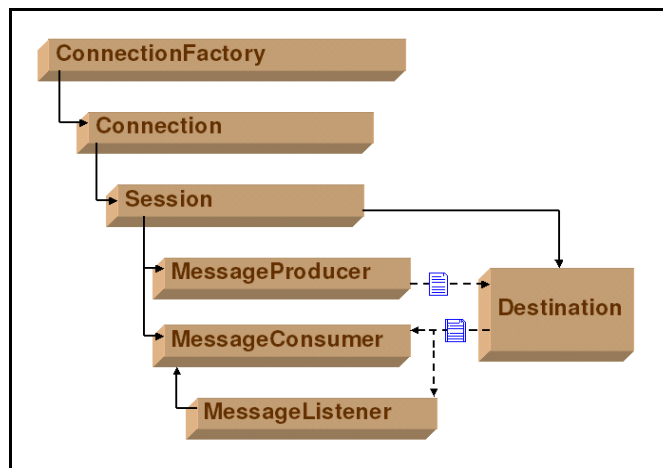


Figure 26. Generic Messaging Model

Message producers are established in one of the messaging domains by creating an appropriate `ConnectionFactory` then creating a `Connections`, then `Sessions` on the `Connections`.

The `Message Producers` send messages to a `Destination` on a message server. `Message Consumers` get messages from a `Destination` by implementing asynchronous `MessageListeners` or doing synchronous receives.

Message Ordering and Reliability

The scope of services available in a loosely coupled messaging structure presents a rich set of factors that impact sequence of messages actually delivered to consumers. Message ordering and redelivery both contribute to reliable message delivery.

General messaging services are impacted by many uncontrollable environmental factors from latency and machine outages to internal factors such as related applications not accepting data types, values, poorly formed XML data, and data payloads. Message delivery is distinctly non-linear.

Message ordering and reliability common to all messaging domains are described in this chapter. See also the `Pub/Sub` and `PTP` chapters for details about message ordering and reliability within those domains.

Messages can be delivered with a range of options to modify message ordering and invoke features that improve reliability:

- The producer can set the **time-to-live** of the message so that obsolete messages can expire. If message **A** is set at one minute, message **B** at five seconds, and message **C** at one hour, after three minutes with no deliveries only message **C** will still exist. Ordering is maintained while expiration is a user-defined value.
- The producer can set the **delivery mode** of messages so that the message server confirms persistent storage of the message before acknowledgement is and the message priority. In the event of a message server failure, a message that the message server acknowledged before it was persisted might be lost.
- The producer can set the **priority** of a message so that the message server can take efforts to position a more recent message before an older one.

- The producer uses a synchronous process to put the message on the message server's message store; when it is released, the message is **acknowledged** as delivered to its interim destination.
- The consumer can use **listeners** to get messages as they are made available.
- Messages sent in the **NON_PERSISTENT** delivery mode can arrive prior to messages that are **PERSISTENT**.
- The consumer starts a session by expressing its preferred **acknowledgement** technique—transactional or not, explicit or implicit.
- Connections can be monitored and, when broken, techniques can automatically attempt to **reconnect**.
- Message senders in the Internet environment are not guaranteed consistent communication times. Transmission **latencies** can cause messages to be actually produced before other messages. As a result two messages from two sessions are not required—and cannot be reliably expected—to be in any specific sequence.

Destinations

Destinations are objects that provide the producer, message server, and consumer with a context for delivery of messages. Destinations can be JMS Administered Objects (static objects under administrative control), dynamic objects created as needed (topics only), or temporary objects created for very limited use. The destination name is a string of any `java.lang.String` length.

For topics, SonicMQ provides extended management and security with **hierarchical name spaces**; for example, `jms.samples.chat`.

Destination names can be any set of characters with a few reservations:

- `.` (period) delimits hierarchical nodes, particularly for topics. See [Chapter 9, “Hierarchical Name Spaces,”](#) for more information.
- `*` (asterisk) and `#` (pound) are used as template characters. These are stored for durable subscriptions and, by managers, for access control lists. The stored characters are applied as wildcards when implemented. See [Chapter 9, “Hierarchical Name Spaces,”](#) for more information.

- `$` (dollarsign) and the strings `$_SYS` and `$_SYS` are administrative topics. See the *SonicMQ Installation and Administration Guide* for more information.
- `:` (colon) delimits the routing node name and a remote message server in Dynamic Routing Architecture deployments.
- `::` (double_colon) delimits a routing node name and a queue name in Dynamic Routing Architecture deployments. See the *SonicMQ Deployment Guide* for more information.

You can programmatically store and retrieve defined topics. SonicMQ lets you store topic names in JNDI or a simple file store and then reference the object indirectly (by name) in some context. See [Chapter 12, “Lookup of Administered Objects,”](#) for more information.

Steps in Message Production

Every time a Pub/Sub session wants to send a message to a topic, it must create a producer in the session for the selected destination. The only exception is when you intend to establish an **unbound** destination—a **null** destination name that, for example, enables the `QueueRequestor` to bind to that queue space.

Producing a Pub/Sub message within a connected session is presented in six steps:

1. Create the publisher session.
2. Create the publisher to the topic.
3. Create the message and setting its content.
4. Set message header fields.
5. Set message properties.
6. Publish the message.

Create the Topic Publisher on the `PublisherSession` Thread

After the connection is established, a session that will be reserved for publisher activities is created:

```
pubSession = connect.createTopicSession  
(false, javax.jms.Session.AUTO_ACKNOWLEDGE);
```

Create the Producer on the Producer Session Thread

In the `pubSession`, the static variable `APP_TOPIC` that was assigned `"jms.sample.chat"` is set up as the working topic and a publisher is associated with it:

```
j avax. j ms. Topic topic = pubSession. createTopic (APP_TOPIC);
publ i sher = pubSession. createPubl i sher(topic);
```

Create the Message Type and Set Its Body

A text message, constructed from the standard input (the keyboard), is read in when the `readLine` is activated. A new `SonicMQ TextMessage` is created and the text is set into it, prepended in the sample by the username, a colon and a space.

```
String s = stdin. readLine();
j avax. j ms. TextMessage msg = pubSession. createTextMessage();
msg. setText( username + ": " + s );
```

If user `Sal es` enters `"Hel lo."`, the message content is `"Sal es: Hel lo."`

Set Message Header Fields

To change header fields, use the set methods for message header fields that are available for change:

```
setJMSType("Central Files")
```

Note that some header field set methods exist (such as `setJMSMessageID` and `setJMSTimestamp`) yet whatever you assign is overwritten at the time the message is produced.

The header fields that are named and typed yet available for assignment are:

- `JMSCorrelationID`, reserved for message matching functions
- `JMSReplyTo`, reserved for request reply information
- `JMSType`, available for general use

Set the Message Properties

Use the set methods for the data type of a property and then supply the property name and its value of the declared type. Generically:

```
set[ type]Property( String name, String value)
```

For example:

```
setLongProperty("OurInfo_AuditTrail", "6789")
```

Produce the Message

When the message is assigned its attributes (header fields and properties) and its payload, the message is ready to be produced. In its simplest form the producer method for a publisher is:

```
publisher.publish(msg);
```

The form of `publish` used in the `DurableChat` sets three important message parameters at the moment the `publish` is executed:

```
private static final long MESSAGE_LIFESPAN = 1800000;
publisher.publish(msg,
    javax.jms.DeliveryMode.PERSISTENT,
    javax.jms.Message.DEFAULT_PRIORITY,
    MESSAGE_LIFESPAN);
```

The message production method passes along either the default values or the entered values for:

- `JMSDeliveryMode` is `[NON_PERSISTENT|PERSISTENT|NON_PERSISTENT_ASYNC]`
- `JMSPriority` is `[0..9]` where 0 is lowest, 9 is highest, 4 is the default.
- `timeToLive`, the message lifespan that will calculate the `JMSExpiration`, is `[0..n]` where 0 is “forever” and any other positive value `n` is in milliseconds.

The message producer method assigns—and overwriting, if previously assigned—data to the following header fields:

- `JMSDestination`, the producer’s current destination
- `JMSTimestamp`, based on the producer’s system clock
- `JMSMessageID`, based on the algorithm run on the producer’s system
- `JMSExpiration`, based on the producer’s system clock plus the `timeToLive`

The release of the synchronous block by the message server returns only a boolean indicating whether the message production completed successfully.

Important While the `JMSEExpiration` is calculated from the client system clock at the time of the send, it is enforced on the message server's clock. To accommodate variances between client and server clocks, the message server adjusts the message expiration to its clock. When the message is forwarded to another message server, the remaining `timeToLive` value (expiration minus current message server GMT time) is forwarded. The time that elapses until the first packet of the message in transit is received is effectively ignored.

Message Management by the Message Server

A message at a destination behaves according to the parameters of the message send (PTP) or publ i sh (Pub/Sub) event. [Table 11](#) lists those parameters and how those parameters tell the message server how to handle the message.

Table 11. How Message Producer Parameters Influence the Message Server

<i>Producer Parameter</i>	<i>How the parameter is treated by the message server</i>
<i>del i veryMode</i>	<p>del i veryMode = PERSI STENT</p> <p>Store the message in the message server’s message log in case of impending failure. Acknowledge the producer only after logging the message.</p> <p>del i veryMode = NON_PERSI STENT</p> <p>If the message is enqueued or stored for a durable subscriber on a message server that shuts down, the message is volatile.</p> <p>del i veryMode = NON_PERSI STENT_ASYNC</p> <p>Message publisher methods do not expect any acknowledgement whatsoever. This delivery mode is often appropriate for “blasting” published data such as current stock market prices.</p> <p>NOTE: A message’s deliveryMode is effective throughout its lifespan. If a NON_PERSI STENT message is enqueued (PTP) or stored for a durable subscriber (Pub/Sub) on a message server that shuts down, the message is volatile. This behavior stays with a message throughout its travels in a dynamic queue routing deployment, and even applies in the dead message queue.</p>
<i>pri ori ty</i>	<p>pri ori ty = 0 . . . 9</p> <p>When there are several messages for a receiver that is awaiting delivery, higher priority messages (5 through 9) can move toward the front of the FIFO list. While there are circumstances where this is desirable, more often keeping a smooth FIFO flow is preferable.</p>
<i>ti meToLi ve</i>	<p>ti meToLi ve = <non-negative long i nteger value></p> <p>Number of milliseconds added to the GMT time of the client when the message is produced to determine the JMSExpi rati on date-time of the message. If the <i>ti meToLi ve</i> is 0, the expiration date-time is also 0, the indication that the message is intended never to expire.</p> <p>The <i>ti meToLi ve</i> feature ensures eventual delivery but can result in out-of-date deliverables when queues are not purged and when durable subscriptions are not formally unsubscribed.</p>

Message Listeners, Receivers, and Selectors

Topic subscribers do not automatically get messages. Having an active session where an application subscribes to a topic does not result in the message getting delivered to the application. You must use an asynchronous listener or a synchronous message receiver.

Message Listeners

A message listener is invoked to initiate asynchronous monitoring of the session thread for consumer messages:

```
setMessageListener(MessageListener listener)
```

where *listener* is the message listener to associate with this session.

The listener is often assigned just after creating the destination consumer from the session, so that the listener is bound to the destination to which a consumer was just created, for example:

```
javax.jms.QueueReceiver receiver =  
    session.createReceiver(queue, username);  
receiver.setMessageListener(this);
```

and:

```
javax.jms.TopicSubscriber subscriber =  
    session.createSubscriber(topic, username);  
subscriber.setMessageListener(this);
```

As a result, asynchronous message receipt becomes exclusive for the session.

Note Message sending is not limited when message listeners are in use. Sending is always synchronous.

Message Receiver

The receiver methods are synchronous calls to fetch messages. The different methods manage the potential block by either not waiting if there are no messages or timing out after a specified period.

Receive

To receive the next message produced for the consumer, use the method:

```
Message receive()
```

This call blocks indefinitely until a message is produced. When a `receive` method is called in a **transacted** session, the message remains with the consumer until the transaction commits. The return value is the next message produced for this consumer. If a session is closed while blocking, the return is **null**.

Receive with Timeout

To receive the next message within a specified time interval and cause a timeout when the interval has elapsed:

```
Message receive(long timeout)
```

where *timeout* is the timeout value (*in milliseconds*)

This call blocks until either a message arrives or the timeout expires. The return value is the next message produced for this consumer, or **null** if one is not available.

Receive No Wait

To receive the next available message immediately or instantly timeout:

```
Message receiveNoWait()
```

The `receiveNoWait` method receives the next message if one is available. The return value is the next message produced for this consumer, or **null** if one is not available.

Note The `ReceiveNoWait` method is unlikely to provide effective message consumption in the Pub/Sub paradigm. The no-wait concept is useful for durable subscriptions, but is unlikely to produce results for normal subscriptions.

The method is very useful in the PTP paradigm where messages wait on a static queue.

Message Selector

While some messaging applications expect to get every message produced to a destination, there are techniques that can reduce the flow of irrelevant messages to a message consumer:

- **Subscription to hierarchical name spaces (Pub/Sub)** — SonicMQ’s hierarchical name spaces let subscribers point to content nodes (and, optionally, to sets of relevant subordinate nodes) to focus publishers into meaningful spaces. For more information, see [Chapter 9, “Hierarchical Name Spaces.”](#)
- **Message filtering within a topic** — JMS defines a syntax that is a subset of SQL-92 conditional expressions that allows a subscriber to filter and categorize messages in the message header and properties based on specified criteria. Because the SonicMQ implementation handles the work, the application and its communication links are more efficient and consume less bandwidth. Message selectors do not access the message body. [Table 12](#), [Table 13](#), and [Table 14](#) summarize the selector syntax presented in the JMS specification and implemented in SonicMQ. Although SQL supports arithmetic operations, JMS message selectors do not. SQL comments are not supported.

Message Selector Syntax

A message selector is a java.lang.String that is evaluated left to right within precedence level. You can use parentheses to change this order. A message selector string can contain combinations of the following elements to comprise an expression:

- **Literals and Indefinites** (See [Table 12.](#))
- **Operators and Expressions** (See [Table 13.](#))
- **Comparison tests** See ([Table 14.](#))
- **Parentheses** control the evaluation of an expression.
- **Whitespace** (spaces, horizontal tabs, form feeds, and line terminators) are evaluated in the same way as in Java.

For example, the following message selector might be set up on a `Bidders` topic to retrieve only high-priority quotes that are requesting a reply:

```
"Priority > 7 AND Form = 'Bid' AND Amount is NOT NULL"
```

Table 12. Literal and Identifier Syntax in Message Selectors

<i>Selector</i>	<i>Element</i>	<i>Format and Requirements</i>	<i>Constraints</i>	<i>Example</i>
Literals	String literals	Zero or more characters enclosed in single quotes.		'sal es'
	Exact numeric literals	Numeric long integer values, signed or unsigned.		57 -957 +62
	Approximate numeric literals	Numeric double values in scientific notation.		7E3 -57.9E2
		Numeric double values with a decimal, signed or unsigned.		7. -95.7 +6.2
	Boolean literals	true or false		true

Table 12. Literal and Identifier Syntax in Message Selectors

Selector	Element	Format and Requirements	Constraints	Example
Identifiers	All	A case-sensitive character sequence that must begin with a Java-identifier start character. All following characters must be Java-identifier part characters.	Cannot be null , true , false , NOT , AND , OR , BETWEEN , LIKE , IN , or IS .	JMSType, JMSXState, JMS_Li nks, PSC_Li nk
	Message header field references	JMSDel i veryMode, JMSPri ori ty, JMSMessageI D, JMSTi mestamp, JMSCorrel ati onI D, or JMSType.	JMSDel i very Mode, and JMSPri ori ty cannot be null .	JMSType
	JMSX-defined property references	null when a referenced property does not exist.	None	JMSXState
	SonicMQ defined properties			JMS_Soni cMQ _preserve Undel i vered
Application-specific property names <i>(do not start with 'JMS')</i>	Audi t_Team			

Table 13. Operator and Expression Syntax in Message Selectors

Selector	Element	Format and Requirements	Example
Operators	Logical	In precedence order: NOT, AND, OR	a NOT IN ('a1', 'a2') a > 7 OR b = true a > 7 AND b = true
	Comparison	=, >, >=, <, <=, <> (for booleans and Strings: =, <>)	a > 7 b = 'Quote'
	Arithmetic	In precedence order: - Unary + or - - Multiply * or divide / - Add + or subtract -	a > +7 a * 3 a - 3
	Arithmetic range between two expressions	i d BETWEEN e2 AND e3 i d NOT BETWEEN e2 AND e3	a BETWEEN 3 AND 5 a NOT BETWEEN 3 AND 5
Expressions	Selector	Conditional expression that matches when it evaluates to true	((4*3)=(2*6))= true
	Arithmetic	Include: - Pure arithmetic expressions - Arithmetic operations - Identifiers with numeric values - Numeric literals	7*5 a/b 7
	Conditional	Include: - Pure conditional expressions - Comparison operations - Logical operations - Identifiers with Boolean values - Boolean literals (true , false)	7>6 a > 7 OR b = true a = true true

Table 14. Comparison Test Syntax in Message Selectors

Selector	Element	Format and Requirements	Example
Comparison tests	IN	Identifier IN (str1, str2, ...) Identifier NOT IN (str1, str2, ...)	a IN ('AR', 'AP', 'GL') a NOT IN ('PR', 'IN', 'FA')
	LIKE	Identifier LIKE (str1, str2, ...) Identifier NOT LIKE (str1, str2, ...) can be enhanced with pattern values: - Underscore (_) stands for any character - Percent (%) stands for any sequence of characters To explicitly defer the special characters _ and %, precede their entry with the Escape character.	a LIKE 'Fr%d' is true for 'Fred' 'Fron'd and false for 'Fern' a LIKE '_%' ESCAPE '\' true for 'JMS_A' and false for 'JMSPriority'
	null	Identifier IS NULL Identifier IS NOT NULL for: - Header field value - Property value - Existence of a property Refer to SQL-92 semantics or the JMS specification for more about comparisons that involve null values.	a is NULL a is NOT NULL

Comparing Exact and Inexact Values

Comparing an `int` value (an exact numeric literal that uses the Java integer literal syntax) and a `float` value (an approximate literal that uses the Java floating point literal syntax) is allowed.

Type conversion is defined by the rules of Java numeric promotion as described in the Java Language Specification which, in part, declares that:

- Unary conversions are from `byte`, `short`, or `char`, to a value of type `int` by a widening conversion and, otherwise, a unary numeric operand remains as is and is not converted.
- Binary conversions called for by operands on data of numeric types. If either operand is of type `double`, the other is converted to `double`. Otherwise, if either operand is of type `float`, the other is converted to `float`. Otherwise, if either operand is of type `long`, the other is converted to `long`. Otherwise, both operands are converted to type `int`.

Steps in Listening, Receiving and Consuming Messages

Receiving and consuming a Pub/Sub message within a connected session is presented in six steps:

1. Implement the listener or receiver to the destination.
2. Create the consumer and listener for the destination.
3. Handle a received message by:
 - Using `instanceOf` to determine if the message is as expected
 - Handling alternate message types
 - Manipulating or parsing body data
4. Get header fields.
5. Get message properties.
6. Consume the message.

Implement the Message Listener

The standard JMS message listener is implemented:

```
public class Chat
    implements javax.jms.MessageListener
    ...
```

Create the Destination and Consumer, then Listen

After getting the `ConnectionFactory` object for the appropriate messaging model, then establishing a connection and session, the session objects are created:

```
javax.jms.Topic topic = subSession.createTopic(
    "jms.samples.chat");
javax.jms.TopicSubscriber subscriber =
    subSession.createSubscriber(topic);
subscriber.setMessageListener(this);
```

Handle a Received Message

In the Chat sample the message is assumed to be text and is intended for output to the standard output stream:

```
public void onMessage( javax.jms.Message aMessage )
{
    javax.jms.TextMessage textMessage =
        (javax.jms.TextMessage) aMessage;
    String string = textMessage.getText();
    System.out.println( string );
}
```

Special Handling When the Message Type is Uncertain

In the XMLChat sample, the message is tested to determine whether or not it is an instance of `XMLMessage` and then handled appropriately:

```
public void onMessage( javax.jms.Message aMessage ) {
    if (aMessage instanceof progress.message.client.XMLMessage){
        ... see Parsing an XML Message
    }else{ // Cast the message as a text message and display it.
        javax.jms.TextMessage textMessage =
            (javax.jms.TextMessage) aMessage;
        System.out.println( "[TextMessage] "
            + textMessage.getText() );
    }
}
```

Parse an XML Message and Extracting Data from Fields

```
// Cast the message as an XML message.
```

```
progress.message.jclient.XMLMessage xmlMessage =
(progress.message.jclient.XMLMessage) aMessage;
// Get the XML document associated with this message.
org.w3c.dom.Document doc = xmlMessage.getDocument();
// Get the sender and content from the message.
org.w3c.dom.NodeList nodes = null;
nodes = doc.getElementsByTagName("sender");
String sender = (nodes.getLength() > 0) ?
nodes.item(0).getFirstChild().getNodeValue() : "unknown";
nodes = doc.getElementsByTagName("content");
String content = (nodes.getLength() > 0) ?
nodes.item(0).getFirstChild().getNodeValue() : null;
// Show the message.
System.out.println("[XML from '" + sender + "' ] " + content);
// Show the message as a tree.
printDocNodes(doc.getDocumentElement(), 0);
System.out.println();
```

Get Message Header Fields

Use the get methods for Header fields, such as:

```
getJMSMessageID()
```

Get Message Properties

Use the get methods for the data type of a property and then supply the property name and its value of the declared type. When a property requested does not exist in a message, the return value is **null**. Generically:

```
get[type]Property(String)
```

For example, `getIntProperty("OurInfo_AuditTrail")`

Warning This example gets an `int` property that was set with (and stored as) a `long`. Attempting to get a property type that is not the type with which the property was set will force coercion of the value to the declared type. If the conversion is not valid, an exception is thrown. See [Table 10, "Permitted Type Conversions for Message Properties."](#)

Consume the message

The application can pass the data in an accepted message to the business application for which it performs its services. Explicit acknowledgement of the JMS message to the message server could be postponed until the business application acknowledges processing with a transaction or audit trail identifier. This value could be passed back to the producer is a reply was requested.

Reply-to Mechanisms

The typical design pattern for request/reply is—as in this PTP example:

- Make a temporary queue.
- Set the `JMSReplyTo` header to this queue.
- Do a synchronous `QueueSender.receive()` on the message.

The temporary destination can be a queue or a topic. The temporary destination could be structured into a requestor helper class, as shown in [Table 15](#).

Table 15. Reply-To Mechanisms in Both Domains

<i>Reply-To Mechanism</i>	<i>Publish and Subscribe Domain</i>	<i>Point-to-Point Domain</i>
Desti nati on	TemporaryTopi c	TemporaryQueue
Hel per cl ass	Topi cRequestor	QueueRequestor

The `JMSReplyTo` message header field contains the destination where a reply to the current message should be sent. Messages with a `JMSReplyTo` value are typically expecting a response. If the `JMSReplyTo` value is **null**, no reply is expected. A response can be optional, and client code must handle the action. These messages are called **requests**.

A message sent in response to a request is called a **reply**. Message replies often use the `JMSCorrelati onI D` to ensure that replies synchronize with their request. A `JMSCorrelati onI D` would typically contain the `JMSMessageI D` of the request.

Temporary Destinations Managed by a Requestor Helper Class

Under Pub/Sub, the `TopicRequestor` uses the session and topic that were instantiated from the `Session` methods. The code excerpts below are from the `TopicPubSubRequestor` and `Replier` samples. Notice that the code never actually manipulates the `TemporaryTopic` object; instead it uses the helper class `TopicRequestor`.

Requestor Application

```
javax.jms.TopicRequestor requestor =
    new javax.jms.TopicRequestor(session, topic);
javax.jms.Message response = requestor.request(msg);
javax.jms.TextMessage textMessage =
    (javax.jms.TextMessage) response;
```

Replier Application

Synchronous requests leave the originator of a request waiting for a reply. To prevent a requestor from waiting, a well-designed application uses the following flow:

1. Get the message:

```
public void onMessage( javax.jms.Message aMessage)
{
    javax.jms.TextMessage textMessage =
        (javax.jms.TextMessage) aMessage;
    String string = textMessage.getText();
}
```

2. Look for the header specifying `JMSReplyTo`:

```
javax.jms.Topic replyTopic =
    (javax.jms.Topic) aMessage.getHeader(JMSReplyTo);
if (replyTopic != null)...
```

3. Send a reply to the topic specified in `JMSReplyTo`:

```
javax.jms.TextMessage reply = session.createTextMessage();
```

Design for Handling Requests

The final steps taken by the message handler represent good programming style, but they are not required by the design paradigm for JMS requests:

- Set the `JMSCorrelationID`, tying the response back to the original request.
- Use transacted session `commit` so that the request will not be received without the reply being sent, for example:

```
reply.setJMSCorrelationID(aMessage.getJMSMessageID());  
replyer.publish(replyTopic, reply);  
session.commit();
```

Writing a Topic Requestor

The default `TopicRequestor` behavior is to block when waiting for a reply. You can write your own `TopicRequestor` class that will timeout (`receive(long timeout)`) or listen to the temp topic as a `Subscriber`, thereby avoiding the

blocking situation. The `javax.jms.TopicRequestor.java` file, listed below, is a start toward creating your own `TopicRequestor` class.

```
// @(#)TopicRequestor.java 1.9 98/07/08
// Copyright (c) 1997-1998 Sun Microsystems, Inc. All Rights Reserved.
package javax.jms;
public class TopicRequestor {
    TopicSession session;

    // The topic session the topic belongs to.
    Topic topic;

    // The topic to perform the request/reply on.
    TemporaryTopic tempTopic;
    TopicPublisher publisher;
    TopicSubscriber subscriber;

    // Constructor for the TopicRequestor class.
    public TopicRequestor(TopicSession session, Topic topic)
        throws JMSException {
        this.session = session;
        this.topic = topic;
        tempTopic = session.createTemporaryTopic();
        publisher = session.createPublisher(topic);
        subscriber = session.createSubscriber(tempTopic);
    }

    // Send a request and wait for a reply.
    public Message
    request(Message message)
        throws JMSException
    {
        message.setJMSReplyTo(tempTopic);
        publisher.publish(message);
        return(subscriber.receive());
    }

    // Close resources when done.
    public void
    close() throws JMSException {
        tempTopic.delete();
        publisher.close();
        subscriber.close();
        session.close();
    }
}
```

Producers and Consumers in JMS Messaging Domains

The two JMS messaging domains provide naming conventions for their use of these general messaging terms as listed in [Table 16](#).

Table 16. Messaging Subclasses in JMS Messaging

<i>JMS Messaging Class</i>	<i>Point-to-Point Subclass</i>	<i>Publish and Subscribe Subclass</i>
Connecti onFactory	QueueConnecti onFactory	Topi cConnecti onFactory
Connecti on	QueueConnecti on	Topi cConnecti on
Sessi on	QueueSessi on	Topi cSessi on
MessageProducer	QueueSender	Topi cPubl i sher
MessageConsumer	QueueRecei ver	Topi cSubscri ber
Dest i nati on	Queue	Topi c

[Figure 27](#) shows the relationship of the session objects in the JMS domains.

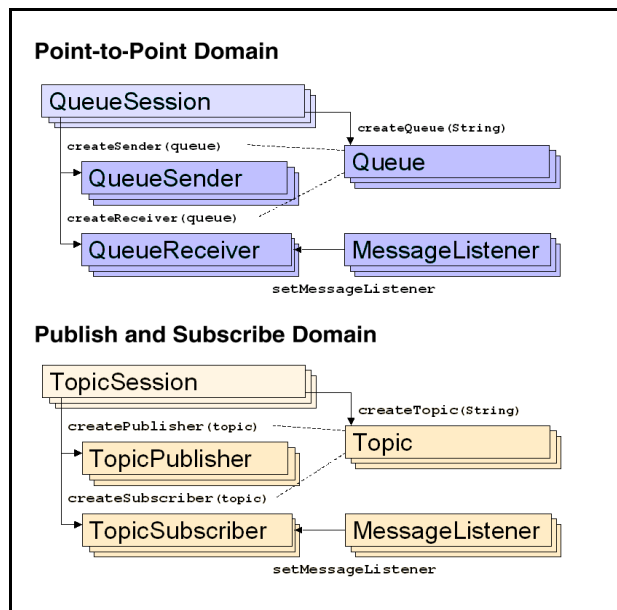


Figure 27. Session Objects in the JMS Domains

Table 17 lists a general messaging functionality that is consistent in both Publish and Subscribe and Point-to-Point messaging.

Table 17. Producer and Consumer Common to Both Messaging Models

<i>javax.jms Interface</i>	<i>Functionality in Either Domain</i>
Destination extended by: Queue, Topic	Destination supports concurrent use
MessageProducer extended by: QueueSender TopicPublisher	Able to send message while connection is stopped Close MessageProducer method Supports message delivery modes PERSISTENT and NON_PERSISTENT Supports message Time-to-Live Support message priority
MessageConsumer extended by: QueueReceiver TopicSubscriber	Close MessageConsumer method Supports MessageSelectors Supports synchronous delivery (receive method) Supports asynchronous delivery (onMessage method) Supports AUTO_ACKNOWLEDGE of messages Supports CLIENT_ACKNOWLEDGE of messages Supports DUPS_OK_ACKNOWLEDGE of messages Supports SINGLE_MESSAGE_ACKNOWLEDGE of messages
Message extended by: TextMessage extended by XMLMessage MapMessage StreamMessage ObjectMessage BytesMessage	Message header fields Message properties Message acknowledgment Message selectors Access to message after being sent for reuse

See Chapter 6, “Point-to-Point Messaging,” and Chapter 8, “Publish and Subscribe Messaging,” for programming concepts and distinguished functionality in each messaging domain.

About Point-to-Point Messaging

In the Point-to-Point (PTP) messaging model, shown in [Figure 28](#), a queue stores messages for as long as they are specified to live, waiting for a receiver. The QueueBrowser mechanism provides a sender with an opportunity to peruse the queue to see how message traffic is moving.

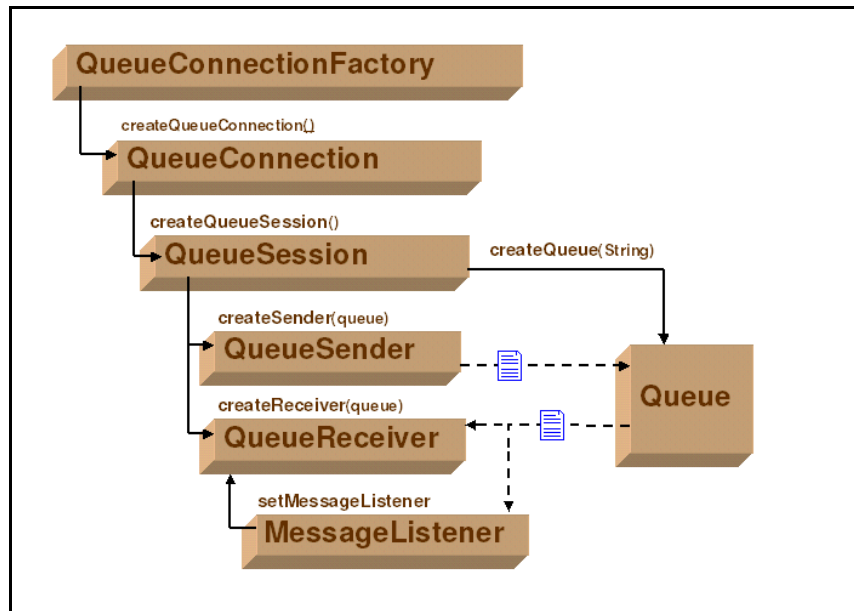


Figure 28. Point-to-Point Messaging Model

Coding Queues, Senders, and Receivers

Queue names must be set up in the message server database by the administrator before they can be used. See the *SonicMQ Installation and Administration Guide* for information about maintaining queues.

Coding Sample

The `QueuePTP` sample `Talk` provides a look at how PTP applications are started and coded. The command that starts the `Talk` application specifies the sending queue and the receiving queue that will be used:

```
java Talk -b <broker:port> -u <user> -p <pwd> -qs <queue> -qr <queue>
```

where:

- `broker:port` points to the message server.
- `user` and `pwd` is the unique user name and its password.
- `-qs queue` is the name of the queue for sending messages.
- `-qr queue` is the name of the queue for receiving messages.

The following segments excerpted from the `Talk` sample show how to create the objects used in PTP communication:

```
// Create a connection. (try/catch)
    javax.jms.QueueConnectionFactory factory;
    factory = (new progress.message.jclient.QueueConnectionFactory
              (broker));
    connect = factory.createQueueConnection (username, password);
    sendSession = connect.createQueueSession
                  (false, javax.jms.Session.AUTO_ACKNOWLEDGE);
    receiveSession =
        connect.createQueueSession
              (false, javax.jms.Session.AUTO_ACKNOWLEDGE);
// Create Sender and Receiver 'Talk' queues. (try/catch)
    if (sQueue != null)
    {
        javax.jms.Queue sendQueue = session.createQueue (sQueue);
        sender = sendSession.createSender (sendQueue);
    }
    if (rQueue != null)
    {
        javax.jms.Queue receiveQueue = receiveSession.createQueue
                                                    (rQueue);
        javax.jms.QueueReceiver qReceiver =
            receiveSession.createReceiver (receiveQueue);
        qReceiver.setMessageListener (this);
        // The 'receive' setup is complete. Start the Connection
        connect.start();
        ...
    } {
```


Message Ordering and Reliability in PTP

The services available in a Point-to-Point messaging structure add other factors to general message ordering and reliability.

Message Ordering

Queued delivery allows several receivers to apply their resources to taking exclusive control of a message and processing that message. As a result, a series of messages might be consumed by several sessions each taking a few messages.

Messages on a queue also have factors that impact the ordering and reliability of messages:

- When a message is put onto a queue, a higher **priority** indicated on the sender method, an active queue receiver might take a newer message off the queue before an older message.
- Queued messages that are not acknowledged are placed back on the queue (**reenqueued**) for delivery to the next qualified receiver. In the interim an older message may have been received by a consumer.
- Queue receivers have a **fetch parameter** that retrieves a number of messages and caches them for processing. If these messages are not processed, they are returned to the queue.

Reliability

Messages on a queue have factors that impact the reliability of messages:

- Message selectors limit the number of messages that a client will receive. Messages could stay on the queue until a receiver either provides a liberal message selector or no message selector at all. A queue might appear empty to a receiver that deselects all the existing messages even though other messages might still be in the queue.
- Message destruction due to administrator action permanently disposes of queued messages.
- Message destruction due to expiration might permanently dispose of a message but the message could—if flagged by the sender—be routed to the

message server's dead message queue where it does not expire. An Administrative application can set up an authorized receiver on the dead message queue to determine whether to recast the message, resend it as is, or discard it.

Note The effects of dynamic routing on message ordering and reliability are discussed at greater length in the scenarios in [Chapter 7, “Dynamic Routing Architecture,”](#) and in the *SonicMQ Deployment Guide*.

Advantages and Constraints in PTP Domains

Consider typical real-world analogies for the basic domains:

- **Point-to-Point** — An available agent for airline check-in takes the person at the front of the line. If there are no agents, you just wait.
- **Publish and Subscribe** — Airport controllers broadcast gate changes to all subscribing airline agents, travel agents, support services, and Web information pages. If you are not connected, you do not get the data.

The concepts of Pub/Sub can be used to simulate PTP functionality by setting up a single administered topic, then giving only one subscriber access to it as a durable subscriber. However, Pub/Sub has restrictions that PTP does not. See [Table 18](#).

Table 18. Advantages of the Point-to-Point Messaging Model

<i>Point-to-Point</i>	<i>Publish and Subscribe</i>
Multiple Receivers — Can set up multiple receivers to take turns at receiving the frontmost message. The message is delivered only once.	Cannot establish another consumer to share the message load. The one message is delivered to every active subscriber.
Queue Browser — Can browse the queue to see what is outstanding and what is frontmost.	Cannot know if the messages are awaiting delivery.
Dead Message Queue — Can express interest in a delivery guarantee and set properties that will channel messages to a special queue when they expire. Also can choose to send an administrative event to a management console.	Cannot know if messages are delivered to any subscriber at all.

Multiple Receivers

Every queue receiver is a message consumer for its associated queue, prepared to receive the next available message. While all the receivers in an active multi-receiver system will expect to get all messages, they do so collectively. A hundred messages to four receivers should result in receivers processing about twenty-five messages each.

Queue receivers do not automatically get messages. Having an active session where an application creates a queue does not result in messages getting delivered to the application. Either an asynchronous listener or a synchronous receiver can be used.

Message Queue Listener

A message listener is invoked to initiate asynchronous monitoring of the session thread for messages on the queue:

```
setMessageListener(MessageListener listener)
```

where *listener* is the message listener to associate with this session.

The listener is often assigned just after creating the queue receiver from the session, so that the listener is bound to the queue to which the receiver was just made:

```
javax.jms.Queue receiverQueue = session.createQueue(rQueue);  
javax.jms.QueueReceiver qReceiver =  
    session.createReceiver(receiverQueue);  
qReceiver.setMessageListener(this);
```

As a result, asynchronous message receipt becomes exclusive for the session. Message sending is not limited when message listeners are in use. Sending is always synchronous.

Message Queue Receiver

The `QueueReceiver` interface provides methods for synchronous calls to fetch messages. The variants manage the potential block by either not waiting if there are no messages or timing out after a specified wait period.

Receive

To receive the next message produced for the queue receiver, use the method:

```
Message receive()
```

This call blocks indefinitely until a message is produced. When a `receive` method is called in a **transacted** session, the message remains with the receiver until the transaction commits. The return value is the next message produced for this receiver. If a session is closed while blocking, the return is **null**.

Receive with Timeout

To receive the next message on the queue within a specified time interval and cause a timeout when the interval has elapsed, use the method:

```
Message receive(long timeout)
```

where *timeout* is the timeout value (in milliseconds)

This call blocks until a message arrives or the timeout expires. The return value is the next message produced for this queue receiver, or null if one is not available.

Receive No Wait

To immediately receive the next available message on the queue or, otherwise, instantly timeout, use the method:

```
Message receiveNoWait()
```

It receives the next message if one is available. The return value is the next message produced for this queue receiver, or null if one is not available.

Prefetch Count and Threshold

SonicMQ extends the standard `QueueReceiver` interface to enable the programmer to set and get parameters of the message receiver that allow performance tuning:

- **Count** — The number of messages that the receiver will take off the queue to buffer locally for consumption and acknowledgement. The default `PrefetchCount` value is 3.
- **Threshold** — The minimum number of messages in the local buffer that will allow a new receiver to append more messages to the buffer. The default `PrefetchThreshold` value is 1.

For example, a threshold value of 2 and a prefetch count of 5 causes the `QueueReceiver` to fetch batches of five messages from the message server whenever the number of messages locally waiting for processing drops below two.

The threshold value cannot be greater than the count value.

setPrefetchCount

```
progress.message.jclient.QueueReceiver.setPrefetchCount(int count)
```

where *count* is the number of messages to prefetch.

When the `PrefetchCount` value is greater than one, the message server can send multiple messages as part of a single `QueueReceiver` request. This can improve performance.

getPrefetchCount

```
progress.message.jclient.QueueReceiver.getPrefetchCount()
```

Returns the `PrefetchCount` positive integer value.

setPrefetchThreshold

```
progress.message.jclient.QueueReceiver.setPrefetchThreshold(int threshold)
```

where *threshold* is the threshold value for prefetching messages.

Setting this to a value greater than zero allows the `QueueReceiver` to always have messages available for processing locally without waiting for a message server interaction. This improves performance.

When the number of messages waiting to be processed by the `QueueReceiver` falls to, or below, the `PrefetchThreshold` number, a new batch of messages will be fetched.

getPrefetchThreshold

```
progress.message.jclient.QueueReceiver.getPrefetchThreshold()
```

Returns the `PrefetchThreshold` positive integer value.

Queue Browsing

A `QueueBrowser` lets a client look at messages in a queue without removing them. Queue browsing is a task that retrieves a cursor in the queue at its current location, forward or backward to the currently-adjacent message. As the queue can be loading and unloading very quickly, browsing is most useful when assessing queue size and rates of growth. Instead of actual message data, the enumeration method can return just the integer count of messages on the queue.

createBrowser

The browser can be created with a session method:

```
session.createBrowser (Queue queue)
```

where *queue* is the queue you want to browse.

createBrowserMessage (MessageSelector)

A message selector string can be added to qualify the messages that are browsed. See [“Message Selector” on page 145](#) for information about selector syntax.

```
session.createBrowser (QueueSession session,  
                      Queue queue,  
                      String messageSelector)
```

where:

- *session* is the queue session in which you want to browse.
- *queue* is the queue you want to browse.
- *messageSelector* is the selector string that qualifies the messages you want to browse.

getMessageSelector

You can get the message selector expression being used with:

```
String getMessageSelector()
```

getEnumeration

You can get an enumeration for browsing the current queue messages in the sequence that messages would be received with:

```
java.util.Enumeration getEnumeration()
```

getQueue

You can get the queue name associated with an active browser with:

```
getQueue()
```

close

Always close resources when they are no longer needed with:

```
close()
```

QueueBrowser Sample

The sample application `QueuePTP\QueueMonitor` uses the Queue Browser to display current queue contents in a Java Window. Some of its code is listed below:

```
// Create a browser on the queue and show the messages waiting in it.
    javax.jms.Queue q = (javax.jms.Queue) theQueues.elementAt(i);
    textArea.append("Browsing queue \"" + q.getQueueName() + "\"\n");
// Create a queue browser
    System.out.print ("Creating QueueBrowser for \"" +
        q.getQueueName() + "\"...");
    javax.jms.QueueBrowser browser = session.createBrowser(q);
    System.out.println ("[done]"); =
    int cnt = 0;
```

```
Enumeration e = browser.getEnumeration();
if(!e.hasMoreElements())
{
    textArea.append("<no messages in queue>");
}
else
{
    while(e.hasMoreElements())
    {
        System.out.print(" --> getting message " +
String.valueOf(++cnt) + "...");
        javax.jms.Message message = (javax.jms.Message)
e.nextElement();
        System.out.println("[ " + message + " ]");
        if (message != null)
        {
            String msgText = getContents (message);
            textArea.append(msgText + "\n");
        }
    }
}
}
```

Handling Undelivered Messages

SonicMQ provides a service whereby an undeliverable message can—if the sender requested the additional service—be taken off its queue and then re-queued on a standard system queue where it will reside until acted on. The dead message queue (DMQ) is a finite data store that is usually managed by message server administrator applications.

You, the programmer, can express interest in trapping items when they are undelivered items. You can set that you want a message to:

- Be placed in the dead message queue when it is discovered to be expired.
- Send a notification, an administrative event.

Note There are several other reasons a message could be undelivered in a dynamic routing deployment. See [Chapter 7, “Dynamic Routing Architecture,”](#) for more about undelivered messages in such an architecture.

Setting Important Messages to Get Saved If They Expire

Important messages should be sent with a **PERSISTENT** delivery mode and flagged to be preserved on expiration or when they cannot be routed

successfully across routing nodes. You could choose to also generate an administrative event. The following code sample shows those settings:

```
// Create a BytesMessage for the payload. Make sure the message  
// is delivered within 2 hours (7,200,000 milliseconds).  
// If expires, send a notification and save the message.  
javax.jms.BytesMessage msg = session.createBytesMessage();  
msg.setBytes(payload);  
// Set 'undelivered' behavior.  
msg.setBooleanProperty(PRESERVE_UNDELIVERED, true);  
msg.setBooleanProperty(NOTIFY_UNDELIVERED, true);  
// Send the message with PERSISTENT, TimeToLive values.  
qsender.send(msg,  
             javax.jms.DeliveryMode.PERSISTENT,  
             javax.jms.Message.DEFAULT_PRIORITY,  
             7200000);
```

Setting Quick Messages to Generate Administrative Notice

Send a small message using high priority, with the expectation that this message will be delivered in ten minutes. Only notification events are needed.

```
// Create a BytesMessage for the payload. Make sure the message  
// is delivered within 10 minutes (600,000 milliseconds).  
// If expires, send a notification.  
javax.jms.BytesMessage msg = session.createBytesMessage();  
msg.setBytes(payload);  
// Set 'undelivered' behavior. Using the property names that  
// are defined as static final Strings in  
// progress.messages.jclient.Constants ensures catching errors.  
msg.setBooleanProperty(NOTIFY_UNDELIVERED, true);  
// Send the message for fast delivery, or not at all.  
qsender.send(msg,  
             javax.jms.DeliveryMode.NON_PERSISTENT,  
             8,           // Expedite at a high priority  
             600000); // 10 minutes
```

Life Cycle of a Guaranteed Message

A message gets sent to the dead message queue only when the application developer declares it important to do so.

Setting the Message to Be Preserved

The application developer can choose to set the property of a message that is about to be sent to declare that the entire message should be preserved if it is undeliverable as follows:

```
msg.setBooleanProperty(PRESERVE_UNDELIVERED, true);
```

You can choose to also generate an administrative event.

Setting the Message to Generate an Administrative Event

You could express an interest in being advised whether or not a message was delivered without needing to preserve the original message. This is distinctly more efficient both in terms of the message traffic density and the requirements of dequeuing undelivered messages. To declare that an administrative event should be generated, set the appropriate message property:

```
msg.setBooleanProperty(NOTIFY_UNDELIVERED, true);
```

Sending the Message

The sending application sends the message metadata and the message payload. It can expect that the message gets delivered to an interested receiver.

Letting the Message Get Delivered or Expire

A message can be acknowledged as delivered to a receiver. If the message is `NON_PERSISTENT`, it is volatile if there is a system outage. If the message is `PERSISTENT`, it will recover from a system outage.

Post-Processing of Expired Message

When a message's expiration time—as marked in the message's `JMSExpiration` header field—has passed, the message server dequeues the message and examines the sender's settings.

Dequeuing only takes place when the messages are reviewed. Inert or low volume queues may have messages that expire but do not become undelivered until a receive or browse mechanism compels the message server to look at the message. Two properties are inspected to see if either or both further processing steps is requested:

- `JMS_SonicMQ_preserveUndelivered` – If **true**, the expired message is transferred to the dead message queue
- `JMS_SonicMQ_notifyUndelivered` — If **true**, the expired message generates an administrative notice.

Processing of Enqueuing Expired Messages

When an expired message is transferred to the dead message queue, it has the reason code `UNDELIVERED_TTL_EXPIRED`. When the message is transferred to the queue `SonicMQ.deadMessage`, the message server adds two properties:

```
JMS_SonicMQ_undeliveredReasonCode = <reason code>
```

```
JMS_SonicMQ_undeliveredTimestamp = <GMT time> [as long]
```

The message retains its original `JMSDestination` header field value. This is unlike all other types of queues where all JMS destinations match the queue definition.

Also the message retains its original `JMSExpiration` header field value. When the message is retrieved from the dead message queue, you can examine its properties including the time at which it was declared undeliverable, an indicator of the time on the system clock where the message expired..

Important Messages in the dead message queue with a **PERSISTENT** delivery mode will not expire. If you have access to administrative functions on a message server, stay alert and dequeue dead messages as soon as possible. Messages with **NON-PERSISTENT** delivery mode are volatile and will perish if the message server restarts.

Sending of Administrative Notification

When an expired message requests administrative notification, a notice is sent with the following information:

- **Undelivered Reason Code.** This is stored in the `JMS_SonicsMQ_undeliveredReasonCode` on the original message. In this case, the message is reason code 1, `UNDELIVERED_TTL_EXPIRED`—undelivered because the message's `timeToLive` expired.
- **MessageID** from `JMSMessageID` on the original message.
- **Destination** from `JMSDestination` of the original message.
- **Timestamp** when the message was handled as a dead message. This is stored in the `JMS_SonicsMQ_undeliveredTimestamp` if the message is saved.
- **Name of the message server** where the notification originated. This is important in clustered message server deployments.
- **Preserved** as set in the `JMS_SonicsMQ_preserveUndelivered` property on the original message. If `true`, the message has been saved in the dead message queue on the server where the message was declared undeliverable.

Programmer Callback for Undelivered Message Notification

Programmatic handling of the undelivered message event uses the management API calls in `progress.message.tools.BrokerManager`. You must create a class that implements the callback for the `brokerUndeliveredMsgNotification` method.

See the Javadoc for the `BrokerManager` class and `IBrokerManagerListener` interface in the `progress.message.tools` package for more information on these calls.

Getting Messages Out of the Dead Message Queue

The following code shows the use of synchronous receives against messages in the DMQ:

```
import progress.message.jclient.Constants;
. . .
// Create a QueueReceiver against the dead message queue.
Session session =
connect.createQueueSession(false, CLIENT_ACKNOWLEDGE);
Queue dmq = session.createQueue ("SonicMQ.deadMessage");
QueueReceiver receiver = session.createQueueReceiver(dmq);
connect.start();

// Empty the dead message queue.
while(true)
{
    Message m = receiver.receive();
    int code =
        m.getIntegerProperty(Constants.UNDELIVERED_REASON_CODE);
    if (code == Constants.UNDELIVERED_TTL_EXPIRED)
    {
        // Handle due to normal timeout.
        . . .
    }
}
```


About Dynamic Routing

This chapter describes some additional Point-to-Point programming techniques that are important when SonicMQ is used with a Dynamic Routing Architecture (DRA) in Business-to-Business (B2B) deployments. A DRA deployment is characterized by the use of **remote queues**, queues identified by a double colon (: :) in their name which must be accessed in special ways. When using remote queues, keep the following points in mind:

- Messages sent to a remote queue that do not reach it must be directed to the Dead Message Queue (DMQ) of the current server or they will be lost.
- A client cannot browse a remote queue, even if the client is connected to the server containing the remote queue.
- A client cannot read a remote queue directly, even if the client is connected to the server containing the remote queue.

To learn about the architecture, functions, and configuration of a SonicMQ Dynamic Routing Architecture, see the *SonicMQ Deployment Guide*.

The behavior of a message under DRA is dependent on several factors:

- Is the queue local or global?
- Does the queue exist on the local message server?
- Is the message server part of a cluster? If it is part of a cluster, is the queue a global queue elsewhere in the routing node?

- How was the queue name referenced when the application created the queue? For example:
 - `<queue>` (non-remote queue)
 - `<routing node name>::<queue>` (remote queue)
 - `::<queue>` (remote queue on local cluster)

Message Behavior on Global and Local Queues

The following scenarios describe the view of a Java client trying to send a message to a global queue, `g`, or a local queue, `l`. Each of these queues exists in some scenarios and does not exist in others. The `Q_NAME` changes in each scenario. Messages are set up to be saved in the Dead Message Queue.

The following code describes what is executed on the client in each scenario:

```
// Static setup
private static String Q_NAME = <Various>

// Set the msg to be preserved in the Dead Message Queue.
msg.setBooleanProperty("JMS_SonicleMQ_preserveUndelivered", true);

// Create a Queue and send the message to this queue.
javax.jms.Queue theQueue = session.createQueue(Q_NAME);
javax.jms.QueueSender sender = session.createQueueSender(null);
sender.send (theQueue, msg);
```


Undelivered Message Reason Codes

The reason names of the SonicMQ associates with undelivered messages are Strings in progress.message.jclient.Constants. [Table 19](#) lists those constants that relate to all queues and those relating only to DRA.

Table 19. Reason Codes for Undelivered Messages

Value	Reason	Scope	Reason Marked as Undeliverable
1	UNDELIVERED_TTL_EXPIRED	All	The current system time on the message server (as GMT) exceeds the message's expiration time (as GMT).
3	UNDELIVERED_ROUTING_INVALID_NODE	DRA	The target routing node in the destination cannot be found in the message server's list of routing connections.
4	UNDELIVERED_ROUTING_INVALID_DESTINATION	DRA	Message received by a message server from a remote routing node has a message destination that does not exist as a global queue in the current routing node.
5	UNDELIVERED_ROUTING_TIMEOUT	DRA	Message received by a message server cannot establish a remote connection to the destination routing node after trying for the specified period of time.
6	UNDELIVERED_ROUTING_INDOUBT	DRA	Message is unacknowledged between message servers, leaving the message in doubt . The message servers try to re-establish the connection and resolve the situation.
7	UNDELIVERED_ROUTING_CONNECTION_AUTHENTICATION_FAILURE	DRA	Routing connection username and password were not authorized at a routing node while connecting to the remote message server.
8	UNDELIVERED_ROUTING_CONNECTION_AUTHORIZATION_FAILURE	DRA	Routing connection username did not have appropriate permissions to connect to the specified routing node.
9	UNDELIVERED_MESSAGE_TOO_LARGE_FOR_QUEUE	All	Message is larger than the maximum size of the queue.

Sending to a Message Server Where Queues Exist

This scenario has the following environment:

- The message server's name is **SonicMQ**.
- Routing node name is **NODE** (`ROUTING_NODE_NAME=NODE`).
- Queue **g** exists as a global queue.
- Queue **l** exists as a local queue.
- The routing table is aware of another routing node named **Portal**.

Figure 29 illustrates the scenario.

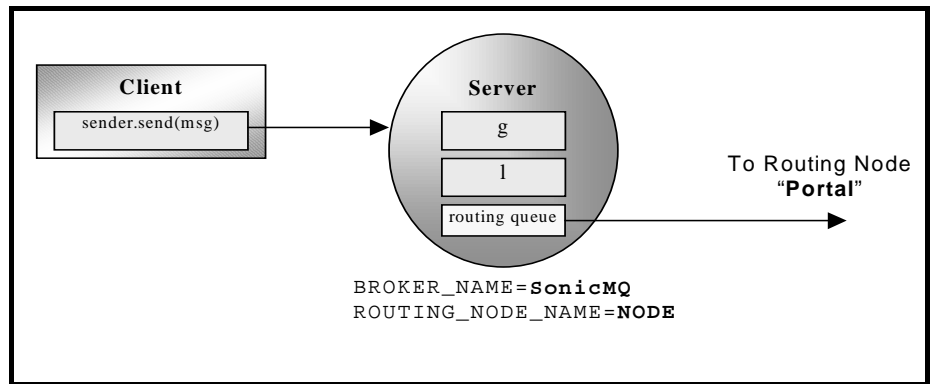


Figure 29. Message Server Where Specified Queues Exist

Table 20 shows the expected behavior for different values of Q_NAME (the queue name used by the client).

Table 20. Routing Behavior on a Server Where Specified Queues Exist

Q_NAME	Behavior	Message Goes To...
l	Send succeeds.	l queue on SonicMQ
NODE::l	Send succeeds. Message goes to routing queue, but cannot be delivered because queue is not global.	Dead Message Queue on SonicMQ Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
::l	Same as NODE::l .	Dead Message Queue on SonicMQ Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
g	Send succeeds.	g queue on SonicMQ
NODE::g	Send succeeds.	g queue on SonicMQ
::g	Same as NODE::g .	g queue on SonicMQ
Portal::appQ	Send succeeds. Message is routed to the Portal Routing Node.	Portal's appQ if it is available; otherwise, Dead Message Queue on Portal Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
Acme::appQ	Send succeeds. However, no routing information exists for the routing node named Acme .	Dead Message Queue on SonicMQ Reason code: UNDELIVERED_ROUTING_INVALID_NODE

Sending to a Message Server Where Queues Do Not Exist

In this scenario, the setup is as follows:

- The message server's name is **SonicMQ**.
- Routing node name is **NODE** (`ROUTING_NODE_NAME=NODE`).
- Queue **g** does not exist (as a global queue).
- Queue **l** does not exist (as a local queue).
- The routing table is aware of another routing node named **Portal**.

Figure 30 illustrates the scenario.

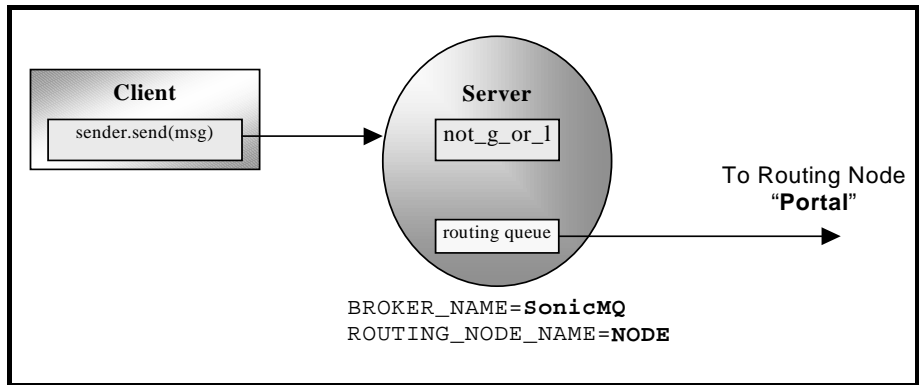


Figure 30. Message Server Where Specified Queues Do Not Exist

Table 21 shows the expected behavior for different values of `O_NAME` (the queue name used by the client).

Table 21. Routing Behavior on Server Where Specified Queues Do Not Exist

Q_NAME	Behavior	Message Goes To...
<code>l</code>	Client gets <code>javax.jms.JMSEException</code> on send.	N/A
<code>NODE:l</code>	Send succeeds. Message goes to routing queue, but cannot be delivered because queue does not exist.	Dead Message Queue on SonicMQ Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
<code>::l</code>	Same as <code>NODE:l</code> .	Dead Message Queue on SonicMQ Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
<code>g</code>	Client gets <code>javax.jms.JMSEException</code> on send.	N/A
<code>NODE:g</code>	Send succeeds. Message goes to routing queue, but cannot be delivered because queue does not exist.	Dead Message Queue on SonicMQ Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
<code>::g</code>	Same as <code>NODE:g</code> .	Dead Message Queue on SonicMQ Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
<code>Portal:appQ</code>	Send succeeds. Message is routed to the Portal routing node.	Portal's appQ if it is available; otherwise, Dead Message Queue on Portal Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
<code>Acme:appQ</code>	Send succeeds. However, no routing information exists for the routing node named Acme .	Dead Message Queue on SonicMQ Reason code: UNDELIVERED_ROUTING_INVALID_NODE

Sending to a Cluster Routing Node With Queues Everywhere

In this scenario, the setup is as follows:

- There are two message servers in the cluster: **Sonic aA** and **Sonic cB**.
- Routing node name is **NODE** (`ROUTING_NODE_NAME=NODE`) for both message servers.
- Queue **g** exists as a global queue on both message servers.
- Queue **l** exists as a local queue on both message servers.
- The client is connected to **Sonic aA**.
- The routing table is aware of another routing node named **Portal**.

Figure 31 illustrates the scenario.

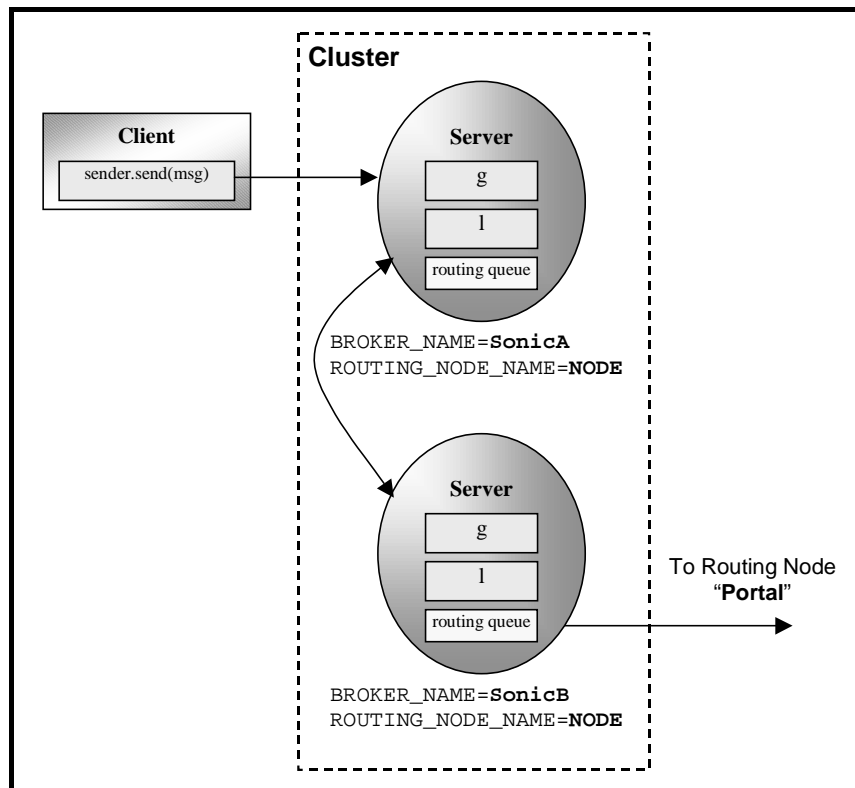


Figure 31. Cluster Routing Node Where Queues Exist On Every Server

Table 22 shows the expected behavior for different values of Q_NAME (the queue name used by the Client).

Table 22. Routing Behavior on a Cluster Node Where Queues Exist on Each Server

Q_NAME	Behavior	Message Goes To...
l	Send succeeds.	l queue on SonicA
NODE::l	Send succeeds. Message goes to routing queue, but cannot be delivered because queue is not global.	Dead Message Queue on SonicA Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
::l	Same as NODE::l	Dead Message Queue on SonicA Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
g	Send succeeds	g queue on SonicA
NODE::g	Send succeeds	g queue on SonicA
::g	Same as NODE::g	g queue on SonicA
Portal::appQ	Send succeeds. Message is routed to the Portal Routing Node	Portal's appQ if it is available; otherwise, Dead Message Queue on Portal Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
Acme::appQ	Send succeeds. However, no routing information exists for the routing node named Acme	Dead Message Queue on SonicA Reason code: UNDELIVERED_ROUTING_INVALID_NODE

Notice that the behavior is identical to that of the non-clustered case because both message servers are identically configured.

Send to a Cluster Routing Node With Queues in One Place

In this example, the setup is as follows:

- There are two message servers in the cluster: **Soni cA** and **Soni cB**
- Routing node name is **NODE** (`ROUTING_NODE_NAME=NODE`) for both message servers.
- Queue, **g**, exists as a global queue, but only on **Soni cB**
- Queue, **l**, exists as a local queue, but only on **Soni cB**
- The Client is connected to **Soni cA**
- The Routing Table is aware of another routing node named **Portal**

Figure 32 illustrates the scenario.

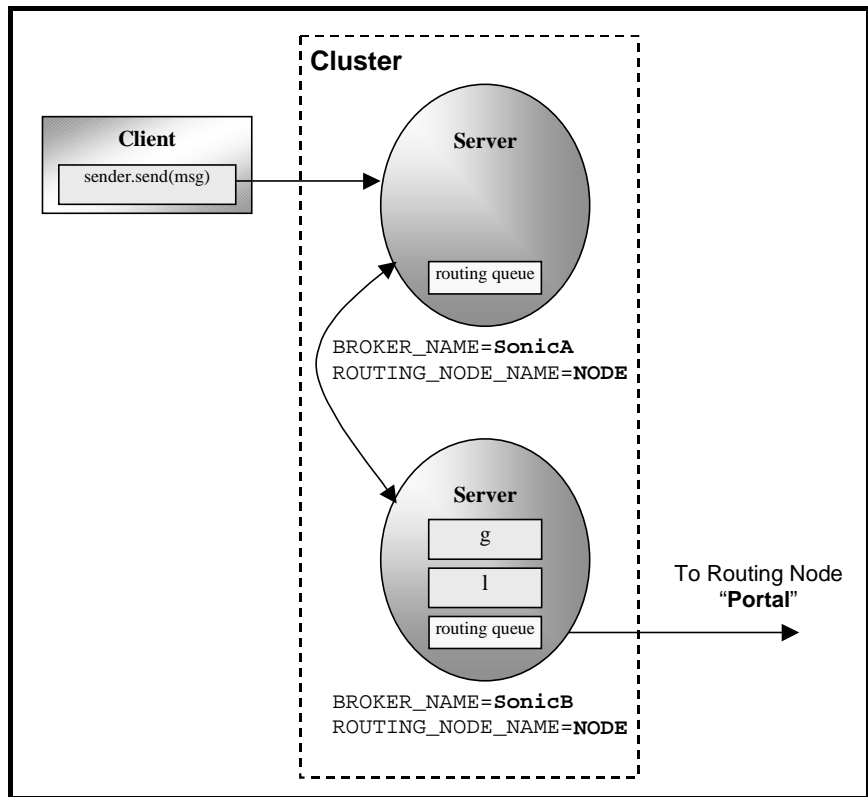


Figure 32. Cluster Routing Node where Queues Exist on Only One Server

Table 23 shows the expected behavior for different values of Q_NAME (the queue name used by the Client).

Table 23. Routing Behavior on Cluster Node Where Queues Exist on Only One Server

Q_NAME	Behavior	Message Goes To...
l	Client gets <code>javax.jms.JMSEException</code> on send.	N/A
NODE:l	Send succeeds. Message goes to routing queue, which routes it to Soni cB . But Soni cB 's routing queue cannot deliver it because the queue is not global.	Dead Message Queue on Soni cA Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
::l	Same as NODE:l	Dead Message Queue on Soni cA Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
g	Client gets <code>javax.jms.JMSEException</code> on send.	N/A
NODE:g	Send succeeds. Message goes to routing queue, which routes it to Soni cB , which delivers it.	g queue on Soni cB
::g	Same as NODE:g	g queue on Soni cB
Portal::appQ	Send succeeds. Message is routed to the Portal Routing Node	Portal 's appQ if it is available; otherwise, Dead Message Queue on Portal Reason code: UNDELIVERED_ROUTING_INVALID_DESTINATION
Acme::appQ	Send succeeds. However, no routing information exists for the routing node named, Acme	Dead Message Queue on Soni cA Reason code: UNDELIVERED_ROUTING_INVALID_NODE

Reply-to Mechanisms for a DRA Application

The typical Reply-to mechanism, described in [“Reply-to Mechanisms” on page 153](#), is appropriate for local message servers. But this solution depends on temporary queues, which are not global and therefore cannot be used in a request/response mechanism in a DRA application. Instead, a client application should implement a synchronous request/reply layer on top of the Dynamic Routing Architecture.

The standard synchronous request/reply design patterns are complicated under DRA because of several issues:

- Creation of unique queues
- Access to queues across B2B security domains

The techniques described here provide an alternate technique for Request/Reply scenario is an example. There is nothing inherent in the Dynamic Routing Architecture that makes this the only way of implementing synchronous request/reply.

Setting Applications to Use Simple Request Messages

Where the standard request/reply mechanism uses the `JMSReplyTo` header field, a DRA application might use the header field instead when the application needs a dialog with other applications to do tasks like price check, inventory status, or credit rating. Typically the dialog is nearly real-time; the application is blocking for a few seconds.

You would use simple request messages to:

- Create request messages with settings for brisk, synchronous requests.
 - Low Quality of Service: Small `NON-PERSISTENT`, unencrypted messages.
 - `TimeToLive` that is explicit and brief.
 - Message expiration is handled by notification. The message is not persisted in a dead message queue.
 - A high `Priority` setting to expedite delivery.
- Implement a retry mechanism in case a request is lost.

Using Specific Shared Reply Queues

Imagine a configuration consisting of a number of nodes called trading partners, each belonging to a different company, and communicating through a special node called a portal. This type of configuration is discussed in detail in the *SonicMQ Deployment Guide*. To protect its position as a necessary intermediary as well as to maintain security, the company that controls the portal must ensure that the trading partners do not communicate directly with one another.

The normal design pattern using a JMS `TemporaryQueue` might not prevent trading partners from knowing about each other's requests. Instead, each trading partner should maintain a specific routing queue, perhaps called **tmpQ**, at its site configured at the portal to ensure security.

Establishing this special purpose queue allows for easier administration. Items on the **tmpQ** queue can be assumed to be transient, and the trading partner can clean up the queue without losing important business documents.

Because many applications at the trading partner might want to simultaneously make requests and get replies, they can share the trading partner's single **tmpQ** queue. By using message selectors, each application can match up requests with targeted replies.

Consider the following code sample for sharing a reply queue:

```
// Create a request
TextMessage m = session.createTextMessage();
m.setJMSReplyTo("acme:tmpQ"); // pseudo-code
m.setText("Inventory Check: #1234");

// Create a unique queue receiver for the reply
// Notice the use of selector
String uniqueID = createUniqueID();
m.setProperty("AppUniqueID", uniqueID);
QueueReceiver qr = session.createQueueReceiver
    ("tmpQ", "AppUniqueID = '" + uniqueID + "'");
// Wait 7 seconds for a reply.
TextMessage rep = qr.receive(m, 7000);
```


About Publish and Subscribe Messaging

Publish and Subscribe (Pub/Sub) is a messaging model, shown in [Figure 33](#), where a message is sent to a **topic**—a content node known by a publisher and active subscribers—so that each subscriber to the content node gets the message. The *one-to-many* model keeps topic publishers independent of the topic subscribers. In fact, publishers could be sending messages to topics where no subscribers exist.

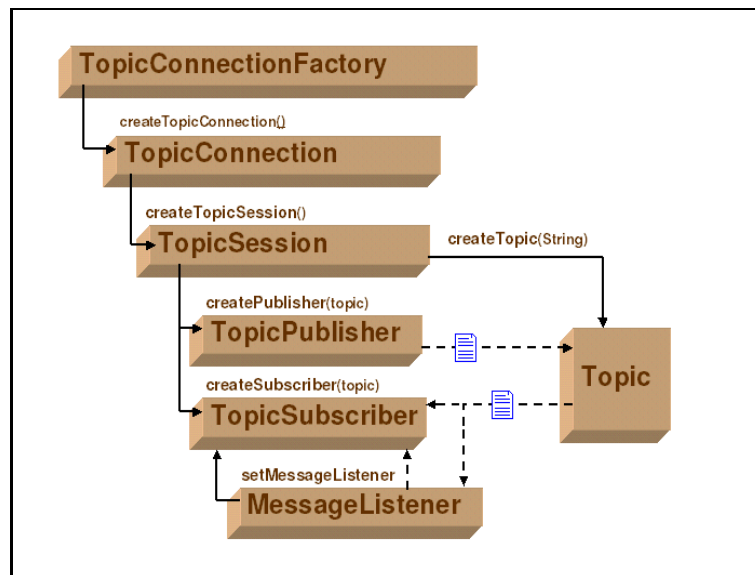


Figure 33. Publish and Subscribe Messaging Model

Mechanisms exist to allow messages to persist for subscribers who have expressed a durable interest in a topic. The characteristics of durable subscriptions are discussed later in this chapter.

See [Chapter 9, “Hierarchical Name Spaces,”](#) for information about how SonicMQ lets applications subscribe to sets of topic content nodes.

Coding Topics, Subscribers, Publishers, and Listeners

The following code excerpted from the Chat sample application shows how to create the objects used in a TopicSession for Pub/Sub communication—the topic, the subscriber, the publisher, and the message:

```
//The topic is defined as a hierarchical topic
String APP_TOPIC = "jms.samples.chat";

//The session method is used to create the topic
javax.jms.Topic topic = session.createTopic(APP_TOPIC);

//The subscriber uses the session method to create a subscriber to it
javax.jms.TopicSubscriber subscriber =
    session.createDurableSubscriber(topic, user);

//The subscriber sets a listener for the topic
subscriber.setMessageListener(this);

//The publisher uses the session method to create a publisher
publisher = session.createPublisher(topic);

// Publish a message to the topic
private void jmsPublish (String aMessage)
{
    try
    {
        javax.jms.TextMessage msg = session.createTextMessage();
        msg.setText(user + ": " + aMessage);
        publisher.publish(msg);
    }
    catch ( javax.jms.JMSEException jmse )
    {
        jmse.printStackTrace();
    }
}
```

Topic

Topics are objects that provide the publisher, message server, and subscriber with a destination for JMS methods. Topics can be static objects under administrative control, dynamic objects created as needed, or temporary objects created for very limited use. The topic name is a string of any `java.lang.String` length.

SonicMQ provides extended topic management and security with **hierarchical name spaces**; for example, `jms.samples.chat`. Some characters and strings are reserved for the features of hierarchical topic structures:

- `.` (period) delimits hierarchical nodes.
- `*` (asterisk) and `#` (pound) are used as template characters.
- `$` (dollarsign) and the strings `$SYS` and `$ISYS` are administrative topics.

See [Chapter 9, “Hierarchical Name Spaces,”](#) for more information.

You can programmatically store and retrieve defined topics. SonicMQ lets you store topic names in JNDI or a simple file store and then reference the object indirectly (by name) in some context. See [Chapter 12, “Lookup of Administered Objects,”](#) for more information.

Publisher

Every time a Pub/Sub session wants to send a message to a topic, it must create a publisher in the session for the selected topic. The only exception is when you intend to establish an **unbound** topic—a **null** topic name that, for example, enables the TopicRequestor to bind to that topic space.

Creating the Publisher

The sample code shows the creation of the topic and the creation of the publisher to that topic:

```
java.jms.Topic topic = session.createTopic("jms.samples.chat");
publisher = session.createPublisher(topic);
```

Figure 34 shows the Explorer view of the parameters and context when you create a publisher to a topic.

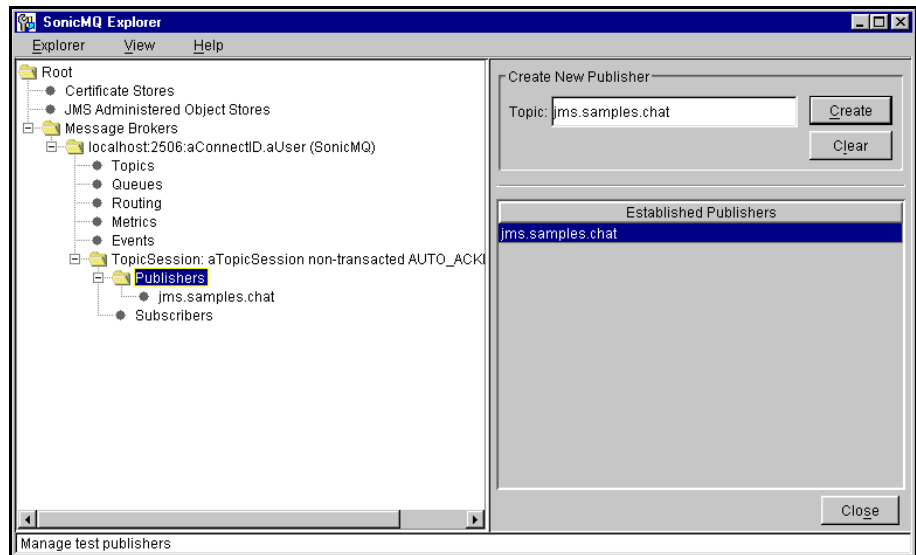


Figure 34. Explorer View of Creating a Publisher

When security is active for topics, the publisher's permission to publish is checked. If the topic is unspecified in the security database, the publisher's right to create new topics is checked.

Creating the Message

The message is created as a session method for the preferred message type. The Chat sample uses the following code to accept input and then create, populate, and publish the input as a text message, prepended with the username of the publisher:

```
while ( true )
{
    String s = stdin.readLine();
    if ( s == null )
        exit();
    else if ( s.length() > 0 )
    {
        javax.jms.TextMessage msg = session.createTextMessage();
        msg.setText( username + ": " + s );
        publisher.publish( msg );
    }
}
```

Publishing to a Topic

The Chat sample simply puts text into the body of the message and accepts every default that is provided for a message. Some message header information is defined by the message server while other header information can be specified by using a `publish` method with a more complex signature, such as the following:

```
publisher.publish(Message message,
                  int    deliverMode,
                  int    priority,
                  long   timeToLive)
```

where:

- *message* is a `javax.jms` message.
- *deliverMode* is `[NON_PERSISTENT|PERSISTENT|NON_PERSISTENT_ASYNC]`.
- *priority* is `[0..9]` where 0 is lowest and 9 is highest.
- *timeToLive* is `[0..n]` where 0 is “forever” and any other positive value *n* is in milliseconds.

Figure 35 shows the Explorer view of the header fields in a text message.

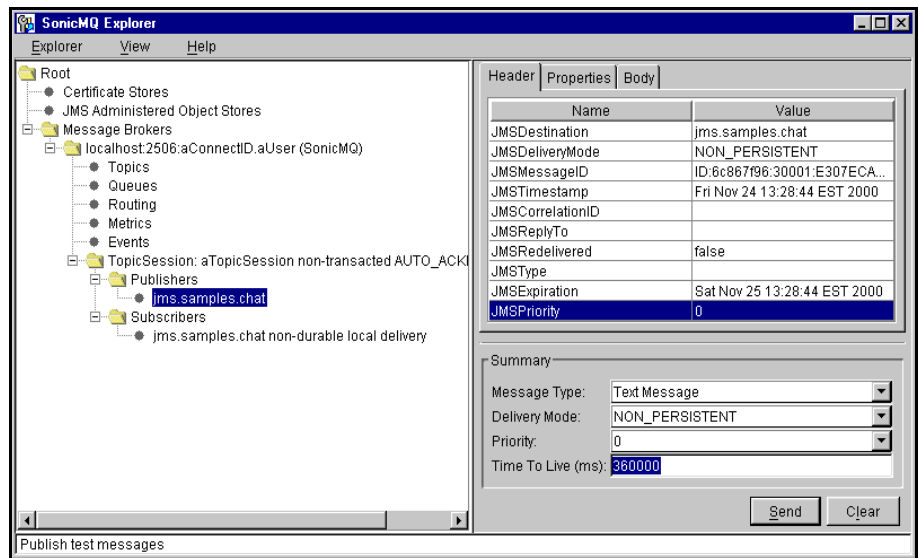


Figure 35. Explorer View of a Message Header Fields After Publishing

Subscriber

A topic subscriber is a message consumer that receives messages when it is active and has specified that it has an interest in a topic. Figure 36 shows the Explorer view of the parameters and context when subscribing to a topic.

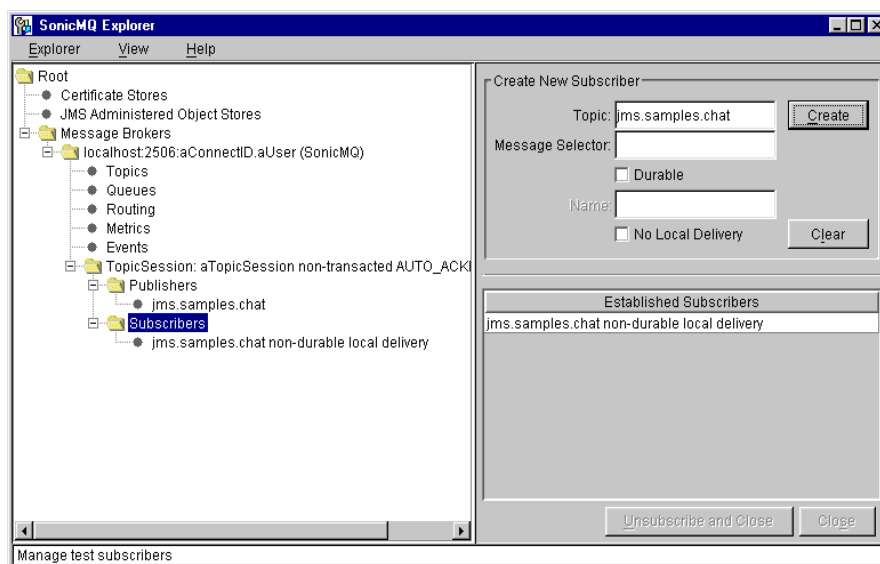


Figure 36. Explorer View of Subscribing to a Topic

The entries describe the parameters of the non-durable subscribe method:

```
Topic cSubscriber createSubscriber (Topic topic,
                                     String messageSelector,
                                     boolean noLocal)
```

where:

- *topic* is a string that specifies the name of a topic.
- *messageSelector* is a string that defines selection criteria.
- *noLocal* is a Boolean where **true** sets the option not to receive messages from subscribed topics that were published locally.

Multiple subscribers in a session could have overlapping subscriptions defined in their message selectors and hierarchical topics. In this case, all subscribers in the session would get the message delivered.

Durable Subscriber

Creating a topic subscription as **durable** expresses that the client wants to receive all the messages published on a topic even if the client connection is not active. The message server notes the durable subscription and ensures that all messages from the topic's publishers are retained until they are either acknowledged by the durable subscriber or the messages have expired. The entries describe the parameters of the `createDurableSubscriber` method:

```
TopicSubscriber createDurableSubscriber  
                (Topic topic,  
                 String subscriptionName,  
                 String messageSelector,  
                 boolean noLocal)
```

where:

- *topic* is a string that specifies the name of a topic.
- *subscriptionName* is a string of arbitrary alphanumeric text and any symbols except “.”, “*”, “#”, and “\$”. The subscription name identifies this unique subscription. It is combined with the user name and the client identifier to define the durable interest. A typical value for the subscription name is a descriptor for the message selector. For example, a durable subscription for messages where the priority value is greater than 7 might have the subscription name `HighPriority`.
- *messageSelector* is a string that defines selection criteria.
- *noLocal* is a Boolean where `true` sets the option to not receive messages from subscribed topics that were published locally.

Durable Subscriptions Not Allowed for Temporary Topics

A durable subscription is not allowed for a temporary topic. An attempt to create a `DurableSubscriber` on a `TempTopic` will throw an exception.

Unsubscribing from a Durable Subscription

While you can stop listening to a topic, there is message server overhead expended when trying to deliver messages to subscribers, especially when the messages might be persistent and the subscribers durable. The `unsubscribe` method unsubscribes a durable subscription that has been created by a client.

This method deletes the state maintained on behalf of the subscriber by its message message server:

```
unsubscribe(String name)
```

where *name* is the name used to identify this subscription.

If you unsubscribe to a durable subscription with undelivered messages and then re-establish a durable subscription to the same topic with the same name, undelivered messages that have not expired for the previous subscription will be delivered to the new durable subscription.

Unsubscribing to Durable Subscription Requires Inactive Subscriber

An **inactive durable subscription** is a durable subscription that exists but does not currently have a message consumer subscribed to it. A `DurableSubscriber` must be inactive before using the `unsubscribe()` method on that durable subscription.

An error will occur when a client tries to delete a durable subscription:

- While it has an active `TopicSubscriber` for it
- While a message received by it is part of a current transaction
- While a message received by it has not been acknowledged

Message Ordering and Reliability

The services available in a Pub/Sub messaging model add other services to message ordering and reliability.

General Services

Asynchronous message delivery lets messages be delivered with a range of options to assure an appropriate quality of service:

- The producer can set the life span of the message, the delivery mode, and the message priority.
- The message server will store the message for later delivery and manage both acknowledgement to the producer and acknowledgement from the consumer.
- The consumer can express a durable interest in a topic.

While general services are impacted by many uncontrollable environmental factors from latency to machine outages, there are internal factors that add complexity. Message delivery is distinctly non-linear.

Message ordering and redelivery can both contribute to message delivery that is reliable.

Message Ordering

A predictable sequence of messages is a series of messages that have the same priority from a single publisher in a single session. Even if transacted, the messages are delivered sequentially from the message server to the consumers. The sequence of messages received by a consumer has several other influences in Pub/Sub domains:

- Changing a priority on a message from a publisher can result in a delivery of a high priority message to a newly-activated subscription before an older message.
- Messages from other sessions and other connections are not required to be in specified sequence relative to messages from another session or connection.

- Published messages that are not acknowledged are redelivered to durable subscribers with an indication of the redelivery attempt. As a result a redelivered message could be received after a message that was timestamped later.

Reliability

The assurance that a message will be received by a consumer has several other influences in Pub/Sub domains:

- A publisher never is guaranteed that any subscriber exists for a topic where messages are published.
- Subscriber message selectors limit the number of messages that a client will receive. Regular subscriptions and durable subscriptions with a message selector definition that excludes a message will never get that message.
- Message destruction due to expiration or administrator action (removing a durable subscription) permanently disposes of stored messages.

About Hierarchical Name Spaces

Hierarchical name spaces are a topic-grouping mechanism available with SonicMQ. When you use topics in the Pub/Sub domain, the publisher, message server, and subscriber all adhere to the JMS standards. But SonicMQ extends topic management in a way that adds virtually no overhead when publishing, yet provides faster access, easier filtering, and flexible subscriptions. By delimiting nodes when naming a topic, a hierarchy of contents is created at the message server. This chapter describes how and when to use hierarchical name spaces.

Advantages of Hierarchical Name Spaces

Naming conventions become cumbersome to work with when long strings are passed around as identifiers. SonicMQ offers the ability to use a naming and directory service with the naming and management of topics. As a result, topics are easier to specify and control for clients and are correspondingly faster to manage and control by the message server.

While a topic hierarchy can be flat (linear), a topic hierarchy typically builds from one or more root topics, adding other topics in levels of parent-child relationships to create a hierarchical naming structure.

The SonicMQ administrator can set and monitor security with the same template character devices to assure that the scope of message permissions is appropriate for each user individually and as a member of one or more groups. See the *SonicMQ Installation and Administration Guide* to learn how security can control access to topic name spaces.

In most messaging systems, there is a one-level structure, as shown in [Figure 37](#).

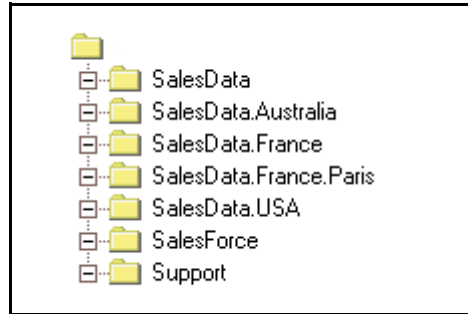


Figure 37. Topic Structure Without Hierarchies

Without hierarchies, many topics are stacked onto one level. When many topics are used, it gets increasingly difficult to maintain access to the naming structure and to denote topic relationships.

Hierarchical name spaces in SonicMQ use a parent-child subordinated folder structure, as shown in [Figure 38](#).

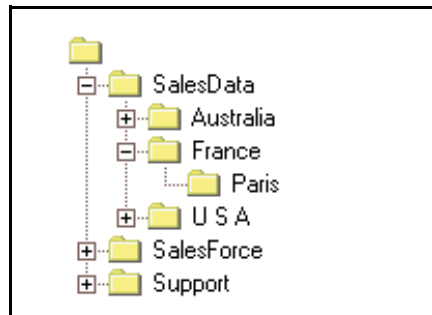


Figure 38. Topic Structure With Hierarchies

With hierarchies, a topic named `SalesData.France.Paris` denotes a content node in a hierarchical structure that can participate in selection mechanisms that refer to its depth in the structure (third-level), the name of the node itself (**Paris**), and its memberships (**Paris** is member of **France** and a member of **SalesData**, among others).

Meaningful names in a topic hierarchy offer many other advantages for message retrieval and security authorization, as discussed later in this chapter.

Publishing a Message to a Topic

Structuring useful topic hierarchies optimizes the management of the hierarchy for the message server and its accessibility by subscribers.

Publishing a message to a topic encourages use of hierarchy delimiters and deprecates the use of a few special characters and topic names.

Topic Notation that Enables Topic Hierarchies

Hierarchical name spaces use the same notation as fully qualified packages and classes: period delimited strings. Security controls whether or not an authenticated user has permission to publish to a topic content node.

See the *SonicMQ Installation and Administration Guide* to learn how security can control publication to topic content nodes.

Reserved Characters when Publishing

Three characters are reserved for special use:

- Delimit the hierarchical nodes with `.` (period). For example, the Chat sample uses the topic name `jms.samples.chat`.
- Do not use `*` (asterisk), `$` (dollarsign), or `#` (pound) in topic names.
- Reserve `$$SYS` and `!$SYS` for administrative topics.

For example, the Chat sample uses the topic name `jms.samples.chat`.

Topic Structure, Syntax, and Semantics

There are few constraints on a topic hierarchy. SonicMQ supports:

- Unlimited number of topics at any content node
- Unlimited depth to the hierarchy (period-separated strings)
- Unlimited length for the name of any topic node, and any topic
- Unlimited length for the complete string that defines a specific node
- Unlimited number of topic hierarchies

Compact, balanced structures always outperform bulky unwieldy hierarchical structures. There are, however, some naming constraints:

- The name must be one or more characters in length with neither leading nor trailing blank space. Embedded spaces are acceptable.
- The topic hierarchies rooted at `$$SYS` and `$$I SYS` are reserved for the message server's system messages.

Note For more information on `$$SYS` and `$$I SYS`, see the *SonicMQ Installation and Administration Guide*.

Topic Syntax and Semantics

The following naming conventions apply to topic naming:

- **Case sensitive** — Topic names are case sensitive (like the Java language). For example, SonicMQ recognizes `ACCOUNTS` and `Accounts` as two different topic names.
- **Spaces in names** — Topic names can include the space character. For example, `accounts payable`. Spaces are treated just like any other character in the topic name.
- **Empty string** — A topic level can be an empty string. For example, `a..c` is a three-level topic name whose middle level is empty. The root node is not a content node, so just an empty string (`" "`) is not a valid topic level for publication.

Note The value `null` indicates an absence of content, or a zero-length string. The Unicode `null` character (`\x0000`) is not a `null` in this convention.

Message Server Management of Topic Hierarchies

Topic hierarchies empower the message server in two significant ways:

- Selection and filtering of topics is, for most purposes, already accomplished. Access to multiple topics is indexed for much faster retrieval than flat naming systems.
- Security that would otherwise be set for each topic individually can be established for a content node and, optionally, its subordinate nodes.

Subscribing to Nodes in the Topic Hierarchy

Subscriptions are created in the JMS standard way with the `Topic` and the `TopicSubscriber` methods. As shown in [Figure 39](#), to get messages published for **U S A Credit**, use the topic name `Credit.U S A`.

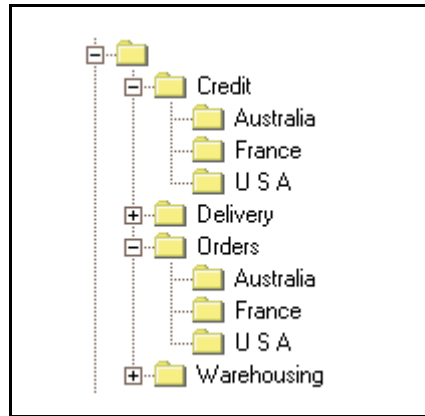


Figure 39. Subscribing to the Topic Credit.U S A

While hierarchical topics enable powerful security and accelerate the retrieval of topics by the message server, SonicMQ topic hierarchies enable unique multiple topic subscriptions, allowing you to:

- Subscribe to many topics quickly.
- Subscribe to topics whose complete name is unknown.
- Traverse topic structures in powerful ways.

When you use topic hierarchies, message selectors—an inherently slow and recurring process—can often be eliminated.

Template Characters

Wildcards are special characters in a sample string that are interpreted when evaluating a set of strings to form a list of qualified names. In this case, however, the special characters are referred to as **template characters** because the entire string and its special characters can be stored for later evaluation by

durable subscriptions and security permissions. The selection of topic names is dynamic, evaluated every time that the topic the time that it is requested.

The `.` (period) delimiter is used together with the `*` (asterisk) and the `#` (pound) template characters when subscriptions are fulfilled. Using these characters avoids having to subscribe to multiple topics and offers benefits to managers who might need to see information or events across several areas. Client applications can only use template characters when subscribing to a set of topics or binding a set of topics to a message handler. Messages must be published on fully specified topic names.

Using template characters is somewhat different from using the usual wild cards as discussed below.

There are two SonicMQ template characters:

- `*` (asterisk) — Selects all topics at this content node.
- `#` (pound) — Selects all topics at this content node and its subordinate hierarchy.

The intent of the template characters is to allow a set of managed topics to exist in a message system in a way that lets subscribers choose broad subscription parameters that will include preferred topics and avoid irrelevant topics.

There are some constraints:

- Unlike shell searches, you cannot qualify a selection, such as `Al pha. B*. Charl i e`. You can use `Al pha. *. Charl i e`. At a content level, a template character precludes using other template characters.
- The `#` symbol can only be used once and only in the last node position. You can use `Al pha. #`, or `*. *. Charl i e. #` or just `#`, but not `#. Beta. Charl i e` or `#. Beta. #`.
- Character replacement, as used in shell searches with the question mark character (`?`), is not allowed.

SonicMQ will deliver a message to more than one message handler if the message's topic matches bindings from multiple handlers.

The content levels in the topic name space consider the root level `""` as level 0.

Using Template Characters in Symmetric Hierarchies

When hierarchical structures are strictly defined, simple templates can be used. For example, the topic hierarchy shown in [Figure 40](#) appears to strictly assign business functions—**Credit**, **Delivery**, **Orders**, and **Warehousing**—to first-level (parent) nodes and a standard set of country names—**Australia**, **France**, **USA**—to second-level (child) nodes.

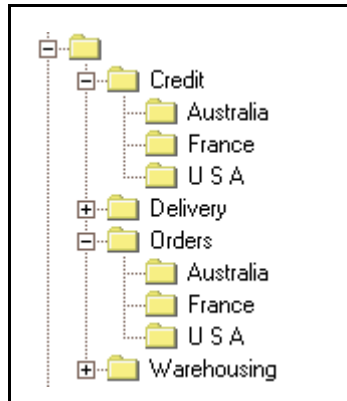


Figure 40. Symmetric Topic Structure

Template Character for All Topics at a Content Level

Using the strict topic hierarchy shown in [Figure 40](#), a client application could subscribe to each of the three topic nodes for **Credit**.

By using a template character, the application can subscribe to all second-level **Credit** topics by subscribing to **Credit.***, a subscription that will deliver messages sent to these destinations:

- **Credit.Australia**
- **Credit.France**
- **Credit.U S A**

Template Character for a Topic at a Content Level

A subscription to the topic expression ***.U S A** in the hierarchy in [Figure 40](#) selects all **U S A** topics at the second level of the hierarchy.

This subscription will deliver messages sent to these destinations:

- **Credit.U S A**
- **Orders.U S A**

Using Template Characters in Asymmetric Topic Hierarchies

When there are several topic levels, as shown in [Figure 41](#), subscribing to all the U S A topics is complicated by an inconsistent topic-naming structure.

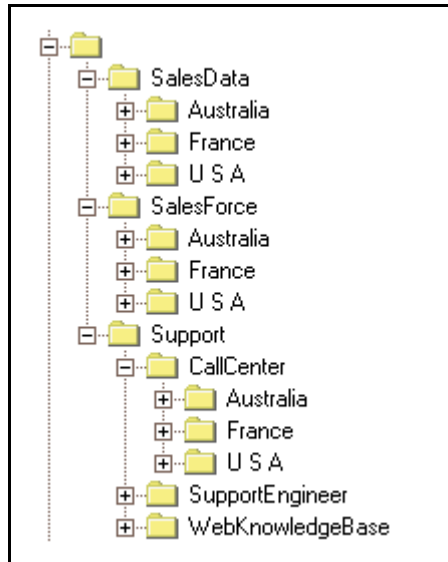


Figure 41. Asymmetric Topic Structure

In this case, the # template character can be used to subscribe to the U S A topic levels in the hierarchy regardless of intervening nodes, such that #.U S A subscribes to topics at these destinations:

- SalesData.U S A
- SalesForce.U S A
- Support.CallCenter.U S A

Without this ability, you would have to subscribe to both *.U S A and *.*.U S A to create the same subscriptions.

Note When you use the "#" template character as the leading character in an expression, you can inadvertently reveal messages in unseen lower levels.

Template Character for Subscribing to All Topics

Subscribing to the topic name # will receive all messages, including the reserved system topics \$SYS and \$I SYS.

The MessageMonitor sample displays all the messages that are published on the message server host by subscribing to #.

Template Character for All Topics Under a Topic Hierarchy

When it is not known how deep the topic structure extends and all subordinate topics are of interest, appending .# extends the subscriptions to all topics at or below that level—for example, Support.# subscribes to:

- Support.Cal I Center
- Support.Cal I Center.Austral ia
- Support.Cal I Center.France
- Support.Cal I Center.U S A
- Support.SupportEngi neer
- Support.WebKnowl edgeBase

plus any subordinate levels below those topic nodes.

Multiple Template Characters in an Expression

Some template characters can be combined in a single expression.

You can:

- Use only one template character at a topic level.
(Support.**.U S A is invalid.)
- Use the pound sign only once in an expression. (#.U S A.# is invalid.)

Examples of multiple template characters in an expression are:

- Use #.U S A.* to subscribe to just the topics at U S A nodes however deep in the topic structure, but not messages at #.U S A.
- Use *.*.U S A.* to subscribe to just the topics at level 4 U S A nodes, but not those at *.*.U S A.

Examples of a Topic Name Space

The hypothetical topic hierarchy shown in Figure 42 has nodes that might represent levels of responsibility in the enterprise.

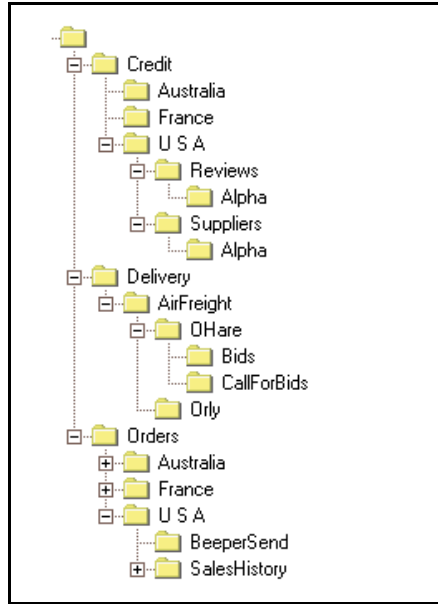


Figure 42. A Sample Hierarchy of Topics

Publishing Messages to a Hierarchical Topic

The publisher produces messages to a single fully qualified topic, such as:

```
static final String MESSAGE_TOPIC = "Credit.U S A. Customers";
```

Business cases where a publisher might use a hierarchical topic are:

- Requests for regular credit updates about suppliers are routed to `Credit.U S A. Suppliers` and use `JMSReplyTo` mechanisms.
- Messages that are sent to credit agencies at secure Internet topics `Credit.U S A. Customers` and `Credit.U S A. Suppliers` should be accessible only by authorized applications.

- Credit agencies can respond to credit requests through the special topic `Credit.U.S.A.Reviews`. Use a `Reviews` topic to get secure responses to credit requests without synchronous blocks.
- As orders are processed through application software, any problems or delays send a message to the appropriate sales force beeper number listed in the application. The message producer uses `Orders.U.S.A.BeeperSend`, attaching the beeper number as the `JMSCorrelationID` or SonicMQ-supplied message property.
- Messages are sent that outline expected shipping needs to topics like `Deliverly.AirFreight.Ohare.CallForBids`.

Subscribing to Sets of Hierarchical Topics

Subscribers to topics can also specify a fully qualified topic:

```
private static final String MESSAGE_TOPIC = "Credit.U.S.A";
```

or use template characters to subscribe to sets of topics:

```
private static final String MESSAGE_TOPIC = "Credit.*";
```

Business cases where a subscriber gains advantage by using template characters to subscribe to hierarchical topics are:

- Accounting subscribes to `Credit.U.S.A.Customers.Reviews` but the auditor subscribes to `Credit.U.S.A.#` to watch all credit activity.
- By listening to `Credit.U.S.A.*.Reviews` the application gets only the U.S.A responses to all types of credit requests without risking synchronous blocks.
- A communications service monitors the message servers at its limited-access read-only topics: `*.U.S.A.BeeperSend` and then executes the beeper activation and download.
- A French affiliate receives all messages that relate to French business by subscribing to `#.France`. This captures:
 - `Orders.France`
 - `Credit.France`
 - `Deliverly.InternationalClearing.France`
- A new bonded carrier in the Chicago area is authorized to monitor `#.Ohare.CallForBids`. Their bids turn around promptly based on the algorithms in their subscriber application.

About the Management API

This chapter presents methods that provide comprehensive programmatic control of administrative tasks. The SonicMQ Management API lets applications perform the tasks that would otherwise require the SonicMQ Explorer GUI interface or the SonicMQ command-line interface, Admin tool, tools that were both constructed using the methods in the exposed Management API.

The Management API facilitates:

- **Message Server Management** — Including methods for checking the status of a message server, subscription to message server events, and information about message servers and server clusters.
- **Destination Management** — Including management of queues and the Quality of Protection (QoP) provided on topics and queues.
- **Access Management** — Including maintenance of users, user groups, and access control by users to destinations.
- **Routing Management** — Including maintenance of routing users, connections, and global destinations.

The exposed Management API is documented in the SonicMQ JavaDoc that you can access through the SonicMQ portal page, `SonicMQ_Help.htm`. From that page you can access the top level of the JavaDoc that opens the `docs/api/progress/message/tools` HTML files.

Using the Management API

An administration client differs from an ordinary (JMS) client in several ways:

- An administration client cannot create or access JMS objects.
- A JMS client cannot access BrokerManager features.
- An administration client and a JMS client cannot share a connection. A Java program that performs JMS actions and uses BrokerManager features requires two connections.

To create an administration client, you must have `broker.jar` in your classpath. Your administration client may require other `jar` files in your classpath as well. Look at `bin\Admi n.bat` (Windows) or `bin/Admi n.sh` (UNIX or Linux) to see the classpath for the Admin Tool.

When programming an administration client, keep the following points in mind:

- The client must be a member of the Administrators group to access security features. If the client is not a member of the Administrators group it can still connect, but it cannot access information.
- You must include an implementation of `IBrokerManagerListener`. However, this implementation can ignore most messages.
- You should call `disconnect()` explicitly at the end of a program.

Samples that Use the Management API

Several samples are included in the SonicMQ installation that let you experience console and application uses of the Administration API. The samples explore portions of the API.

Events

The Events sample provides an application technique that replicates features and functionality in the SonicMQ Explorer and the SonicMQ Admin tool.

The Events sample establishes the BrokerManager and its listener as follows:

```
public class Events
    implements progress.message.tools.IBrokerManagerListener
    ...
    // Create an instance of the BrokerManager.
    m_manager = new progress.message.tools.BrokerManager
                (this, m_broker, m_adminUser, m_adminPassword);
    try
    {
        m_manager.connect();
        m_manager.subscribeToBrokerEvents(m_events);
    }
    ...
```

The notifications that are received are then formatted and displayed:

```
public synchronized void brokerEventNotification
                        (String description)
{
    System.out.println (description);
}
```

The Events sample offers different ways to explore its capabilities of echoing management events:

- Accessing, by default, all events
- Accessing selected events
- Piping the selected events into a text file

Note Events are propagated among all servers of a SonicMQ cluster.

Accessing All Events

This sample procedure opens the management events monitor and then uses a Talk session to fire some events.

► **To start the Events sample:**

1. Open a console window to the BrokerManager\Events folder, then enter:
`.. \. . \SonicMQ Events`

The application defaults to the username **Administrator** and its password **Administrator** as you must have administrator privileges to do the task.

The Events sample starts on the **localhost** port **2506** and displays:

Type **EXIT** to stop listening for BrokerManager Events.

► **Starting and Stopping Talk**

1. Open a console window to the QueuePTP\Talk folder, then enter:
`.. \. . \SonicMQ Talk -u CreditReview`

The Talk sample starts and the Events window displays:

```
>[00/11/23 18:39:50] Connection opened  
[Broker=SonicMQ, User=CreditReview, ConnectID=$CONNECT$3$]
```

2. In the Talk window, press **Ctrl+C** then enter **Y**.

The Talk sample exits and the Events window displays:

```
[00/10/16 17:48:43] Connection dropped  
[Broker=SonicMQ, User=CreditReview, ConnectID=$CONNECT$3$]
```

Accessing Selected Events

The set of events that are accessed can be strictly defined. The events you can choose are the following:

- **connect** — Information about the user and the connectID of a successful connection to a message server.
- **reject** — Information about a connection request that is rejected by a server.
- **drop** — Information about a connection that is lost without the client being disconnected, for example, if the client dies.

- `di sconnect` — Information about the user and the connectID of a successful message server disconnection.
 - `undel i vered` — Information about a message that could not be delivered. Undelivered messages may be enqueued in the dead message queue (DMQ).
 - `dmqstatus` — Indicates that the DMQ has exceeded a defined percentage of its maximum size.
 - `redi rect` — Indicates that a connect attempt has been redirected to another server due to load balancing.
 - `al l` — All of these events. This is the default value.
- **To start the Events sample to display only selected events:**
1. Open a console window to the `BrokerManager\Events` folder, then enter:
`.. \.. \Soni cMQ Events -e connect`
 2. Start then stop the `Tal k` sample.

The `connect` event displays but the `di sconnect` event does not display.

Piping Events Into a Log

You can use standard redirection methods to pipe event records to a disk file.

- **To send event information to a log file:**
1. Open a console window to the `BrokerManager\Events` folder, then enter:
`.. \.. \Soni cMQ Events > LogThi s. txt`
 2. Start then stop the `Tal k` sample.
The events do not display in the console window.
 3. Open the text file `LogThi s. txt`.
The events are recorded as in the following example.

```
Type EXIT to stop l istening for BrokerManager Events.
>[00/11/23 18: 39: 50] Connecti on opened
[Broker=Soni cMQ, User=Credi tRevi ew, ConnectI D=$TMPAPPI D$O$]
[00/11/23 18: 40: 25] Connecti on dropped
[Broker=Soni cMQ, User=Credi tRevi ew, ConnectI D=$TMPAPPI D$O$]
...
```

Metrics

The `Metrics` sample provides an application technique that replicates features and functionality in the SonicMQ Explorer and the SonicMQ Admin tool. The following metrics are provided:

- **Memory Usage** — The number of memory bytes in use by the JVM instance for the message server.
- **Physical Connections** — The current number of socket connections.
- **Msgs Rcvd** — The number of messages received by the server from publishers or senders in the last interval.
- **Msgs Rcvd/sec** — The `Msgs Rcvd` metric as a per second rate.
- **Bytes Rcvd/sec** — The byte-count of the `Msgs Rcvd` as a per second rate.
- **Msgs Dlvd** — The number of messages delivered in the last interval. It counts messages delivered to subscribers in the Pub/Sub domain and messages delivered to queue receivers in the PTP domain.
- **Msgs Dlvd/sec** — The `Msgs Dlvd` metric as a per second rate.

The message server's default interval length for metrics collection is 10 minutes, with a refresh rate of 20 seconds. Every time the message server refreshes, it assesses the metrics. You can append the `-r` parameter and an integer value for your preferred console refresh rate to the command line but the message server's actual refreshes are adjusted in its `broker.ini` file.

➤ **To start the `Metrics` sample:**

1. Open a console window to the `BrokerManager\Metrics` folder, then enter:
`..\..\SonicMQ\Metrics`

The application defaults to message server `-b localhost:2506`.

The username `-u Administrator` and its password `-p Administrator` are defaulted as you must have administrator privileges to do the task.

The `Metrics` sample starts and displays an initialized set of information similar to:

```
Metrics for Broker: SonicMQ
Memory Usage : 3250640
Physical Connections : 1
Msgs Rcvd : 9
Msgs Rcvd/sec : 0
```

```
Bytes Rcvd/sec : 5
Msgs Dlvd : 9
Msgs Dlvd/sec : 0
```

```
Type EXIT to stop polling for BrokerManager Metrics.
>
```

► To run QueueRoundTrip:

1. Open a console window to the QueuePTP\QueueRoundTrip folder, then enter:
`..\..\SonicMQ\QueueRoundTrip -n 100`

The QueueRoundTrip sample runs 100 looped sends and receives.

The **Metrics** window displays information similar to the following:

```
Metrics for Broker: SonicMQ
Memory Usage : 3220616
Physical Connections : 2
Msgs Rcvd : 392
Msgs Rcvd/sec : 81
Bytes Rcvd/sec : 221
Msgs Dlvd : 250
Msgs Dlvd/sec : 90
```

```
Type EXIT to stop polling for BrokerManager Metrics.
>
```

2. In the **Metrics** window, enter **EXIT**.

Piping Metrics Into a Log

You can use standard redirection methods to pipe metric data to a disk file.

► To send metrics data to a log file:

1. Open a console window to the BrokerManager\Metrics folder, then enter:
`..\..\SonicMQ\Metrics > LogThat.txt`

The Metrics sample runs but no data displays in the console window.

2. Start and run any application on the same server to generate meaningful changes into the log file. The events do not display in the console window.
3. In the **Metrics** window, enter **EXIT**.
4. Open the text file LogThat.txt. The metrics are recorded in the text file.

Setup Queues

You can create client application routines that let authorized users set up new queues and the parameters of those queues. This sample application replicates features and functionality in the SonicMQ Explorer and the SonicMQ Admin tool by simply acting on the command line entry to complete its task.

➤ **To set up a queue programmatically:**

1. Open a console window to BrokerManager\SetupQueue folder, then enter:

```
.. \. . \SonicMQ SetupQueues
```

plus the parameters you want to specify:

```
-b <broker: port>      [Default: localhost: 2506]
-u <username>         [Default: Administrator]
-p <password>         [Default: Administrator]
-r <retrieve_extent> [Default: 1200]
-s <save_extent>      [Default: 1400]
-m <maxqueue size>   [Default: 1000]
```

plus switches that set the queue's status:

```
-g global
-e exclusive
```

plus the one or more queues that you want to create with the settings:

```
-q <name1> -q <name2> ...
```

for example:

```
.. \. . \SonicMQ SetupQueues -q NewQueue -m 2000 -g global
```

You can use the SonicMQ Explorer as shown in [Figure 43](#) to display the list of queues and—if you select the option—system queues.

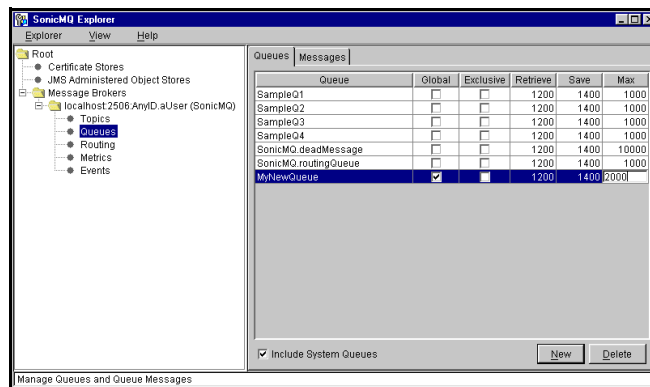


Figure 43. Explorer View of a Newly Created Queue

Show Setup

The ShowSetup sample outputs information about the basic message records in well-formed Admin tool command lines. By flowing output from ShowSetup into a file, the configuration of the message server can be recreated by running the file as an Admin tool script on another message server.

Accessing All Message Server Queue Information

This sample accepts the default server and administrative user to launch an Admin tool script and echo its results into either the console or a file where you have redirected the output.

► **To start the ShowSetup sample:**

- Open a console window to the BrokerManager\ShowSetup folder, then enter:
`.. \. \SonicMQ ShowSetup`

The console displays information similar to the following:

```
// Admin Script to duplicate the setup for broker "SonicMQ".
// This broker is NOT security enabled.

// Connect to the broker.
connect broker localhost Administrator Administrator

// Create queues.
set queue NewQueue global shared 1200,1400,2000
set queue SampleQ3 local shared 1200,1400,1000
set queue SampleQ2 local shared 1200,1400,1000
set queue SampleQ1 local shared 1200,1400,1000
set queue SampleQ4 local shared 1200,1400,1000

// Override properties of system queues.
set queue SonicMQ.routingQueue local shared 1200,1400,1000
set queue SonicMQ.deadMessage local shared 1200,1400,10000

// Create routing connections for "SonicMQ"
// No routing connections defined.

// Close the Admin Shell.
bye
```

Accessing Selected Message Server Queue Information

You can choose the objects that you want reported:

- **queues** — All PTP destinations including system queues.
- **routings** — Routing table information.
- **qops** — Quality of Protection.
- **acls** — Access Control Lists when the security database is active.
- **users** — Users when the security database is active. Passwords are not reported
- **groups** — User groups when the security database is active.
- **groupusers** — Users in each user group when the security database is active.
- **all** — All of the above. This is the default value.

Each option is declared with a `-s` parameter

► To start the ShowSetup sample for specified objects

- Open a console window to the BrokerManager\ShowSetup folder, then enter:
`..\..\SonicMQ ShowSetup -s queues`

The console displays information similar to the following:

```
//  
// Admin Script to duplicate the setup for broker "SonicMQ".  
// This broker is NOT security enabled.  
//  
  
// Connect to the broker.  
connect broker localhost Administrator Administrator  
  
// Create queues.  
set queue NewQueue global shared 1200,1400,2000  
set queue Sample03 local shared 1200,1400,1000  
set queue Sample02 local shared 1200,1400,1000  
set queue Sample01 local shared 1200,1400,1000  
set queue Sample04 local shared 1200,1400,1000  
  
// Override properties of system queues.  
set queue SonicMQ.routingQueue local shared 1200,1400,1000  
set queue SonicMQ.deadMessage local shared 1200,1400,10000  
  
// Close the Admin Shell.  
bye
```

Shutdown

The Shutdown sample provides programmatic access to message server shutdown for authorized users. This replicates the suggested administration function in the Admin tool or the Explorer.

Important In Chapter 2, “Examining the SonicMQ Samples,” the reliable samples suggested the crude **Ctrl+C** technique to emulate unexpected message server shutdown. This action should always be avoided in production.

Instead, call an applications like this one or use the Admin tool or Explorer functions to perform an orderly message server shutdown.

The shutdown process is described in the following code segment from Shutdown.java:

```
...
private void shutdownBroker() throws Exception
{
    // Create an instance of the BrokerManager, and then shut it down.
    m_manager = new progress.message.tools.BrokerManager
        (this, m_broker, m_adminUser, m_adminPassword);
    try
    {
        m_manager.connect();
        m_manager.shutdownBroker();
        // Notify user that we have sent the request (successfully).
        System.out.println
            ("Shutdown request sent to broker \"" + m_broker + "\".");
        m_manager = null;
    }...
}
```

Note that the brokerShutdown request effectively disconnects this client, so you cannot get notification of the shutdown request that was sent.

► **To shut down a server programmatically:**

You could declare the server host, administrator and password but all of them will default to the introductory values when not stated.

- Open a console window to the BrokerManager\Shutdown folder, then enter:
.. \. \SonicMQ Shutdown

► **To shut down a server programmatically with password prompting:**

This variation of the Shutdown sample performs a useful task when the `-p` parameter contains `prompt` instead of the actual password for the username. Under an evaluation setup, the host port is `localhost:2506` with the user `Administrator` and the password `Administrator`.

1. Open a console window to the `BrokerManager\Shutdown` folder, then enter:
`..\..\SonicMQ Shutdown -p prompt`

The window displays: Enter password for user "Administrator">

2. Enter `Administrator` and press Enter.

The window displays:

Shutdown request sent to broker "localhost:2506".

then the application exits.

3. Look at the console window where the message server was running. The following text is displayed:

```
SonicMQ Broker started, now accepting tcp connections on port 2506
...
```

```
Received shutdown request, starting shutdown
```

```
Closing all client connections
```

```
Waiting 30 seconds for threads to shut down
```

```
SonicMQ Broker now exiting...
```

```
Press any key to continue . . .
```

4. When that console window has the focus and you press any key, the console window closes.

Accessing SonicMQ Through ActiveX/COM Clients

About SonicMQ Through ActiveX/COM

SonicMQ provides a component framework that allows JMS objects embedded in applications to communicate with the component framework on a Windows platform.

The SonicMQ interface is packaged for ActiveX/COM so that the objects and methods in the native Java classes are wrapped and the Java events are presented like native ActiveX/COM control events.

SonicMQ clients under ActiveX/COM can have one connection to a message server for each instance of the ActiveX/COM control that is active.

By bridging SonicMQ to ActiveX/COM, Windows developers can:

- **Use familiar tools to make components that interface to SonicMQ** — Microsoft Visual C++, Microsoft Visual Basic, Borland C++, Borland Delphi, Java and others.
- **Run JMS-enabled components in popular applications** — Microsoft Office, Internet Explorer, Lotus® Notes, Lotus SmartSuite®, and more.

Implementation Notes

The SonicMQ connection to ActiveX/COM enables many of the essential functions of loosely coupled messaging:

- `createListener` is available for asynchronous communications with event handlers that provide `onJMSmessage` functionality similar to the JMS native `onMessage`.
- Synchronous receivers can be used.
- Standard JMS message types are supported except for `ObjectMessage`. The SonicMQ `XMLMessage` type is supported.
- `ExceptionHandler`s are available for returning information about JMS Exception events.

The limitations of the ActiveX/COM client interface impose minor constraints:

- `ConnectionFactory`s are not supported through the interface.
- Use of JNDI to find destinations is not supported. However, other file-based store and non-JNDI interfaces to directory services are available.

Requirements for an ActiveX/COM Client

The main elements of the ActiveX/COM client are:

- `activex.jar` in the SonicMQ install directory's `\lib\`.
- Javasoftware JRE v1.2 with JavaBeans Bridge for ActiveX/COM Plug-in
- The SonicMQ TypeLibrary, `Activex.tlb`

See the *SonicMQ Installation and Administration Guide* for more information about installing SonicMQ and its ActiveX/COM Client.

SonicMQ ActiveX/COM Sample

The Visual Basic form, shown in [Figure 44](#) with sample messages, provides access to many of the fundamental procedures of an ActiveX/COM control acting as a client. The sample form is located in the SonicMQ install directory at `samples/ActiveX/Chat/Chat.frm`.

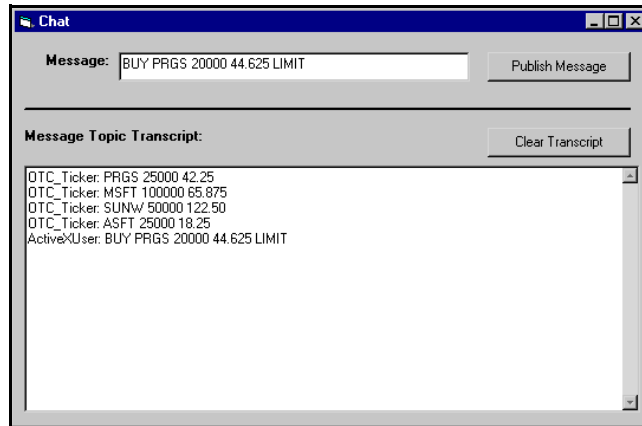


Figure 44. SonicMQ ActiveX/COM Sample, Chat.frm, in Visual Basic

The SonicMQ ActiveX/COM sample form has the following fields and buttons:

- **Message text box** — The entry area for text you want to send as a message.
- **Publish Message button** — The action that publishes the message.
- **Message Topic Transcript text box** — The log of received messages.
- **Clear Transcript button** — Clears the Message Topic Transcript text box.

The form and its code demonstrate how to write a Microsoft Visual Basic Pub/Sub application that uses the SonicMQ ActiveX/COM control. This sample publishes and subscribes to a specified topic. Text you enter is published and then received by all subscribers.

Warning Before you can run the SonicMQ ActiveX/COM sample you must install the SonicMQ ActiveX/COM control. See the *SonicMQ Installation and Administration Guide* for prerequisites, installation, and setup instructions for the SonicMQ ActiveX/COM control.

➤ **To prepare the SonicMQ ActiveX/COM sample:**

When the installation and setup are completed, do the following:

1. Load the project in Visual Basic.
2. Add a reference to the control by choosing **Project > References**.
3. In the Available References list, select the reference `SonicMQ Bean Control`.

Note If you are installing a new version of SonicMQ you may need to reset the path to the control. Be sure the ActiveX/COM jar file that is referenced is the one that is associated with the current version of SonicMQ.

➤ **To modify Chat.frm when security is active:**

If the message server is running with security, the sample source code must be modified to include a valid username and password:

1. In the sample program `Chat.frm` locate the line:
`username = "ActiveXUser"`
2. Change `ActiveXUser` to a username that has been set up in the Access Control List.
3. Change the next line, `password = "password"` to contain the password for the username you specified.
4. Save `Chat.frm`.

For more information, see the comments in `Chat.frm`.

➤ **To set up the message server and console client for the SonicMQ ActiveX/COM sample:**

1. Start the message server (if it is not already running) from the Start menu command: **Start > Programs > Progress SonicMQ > Start Broker**.
2. Open a console window to the install directory folder
`\samples\TopicPubSub\Chat`.
3. Enter:
`..\..\SonicMQ Chat -u Console_Client`

► **To run the SonicMQ ActiveX/COM sample**

The ActiveX/COM sample will use the message server and a console Chat window. The Visual Basic Chat and the console Chat will send each other messages.

1. In the Visual Basic project, choose **File > Make chat.exe**. The path should be into the samples directory `ActiveX/chat`.
2. Choose **Run > Start**. The form runs and its GUI window displays.
3. Type text in the Message text box.
4. Click **Publish Message**. The text you entered displays in the Message Topic Transcript text box. The message also displays in the console client window prefaced by its client name `ActiveXUser`.
5. In the console client window, enter text and then press **Enter**. The text you entered displays in the console window. The message also displays in the Message Topic Transcript text box, prefaced by its client name `ConsoleClient`.
6. The text you entered is retained in the Message text box. Modify or clear the text to send the next message.

Visual Basic Code for the ActiveX/COM Sample

The Visual Basic code that defines the functionality within the form in [Figure 44](#) is detailed in the following listing of `Chat.frm`.

Note The only omission in the code is the GUI form properties definitions.

```
VERSION 5.00
Begin VB.Form Chat
    Caption = "Chat"
    ... ' GUI form properties definitions were omitted here

    Attribute VB_Name = "Chat"
    Attribute VB_GlobalNameSpace = False
    Attribute VB_Creatable = False
    Attribute VB_PredeclaredId = True
    Attribute VB_Exposed = False
```

```
' Copyright (c) 1999, Progress Software Corporation - All Rights Reserved
' Sample Application
' Using the SonicMQ ActiveX control to Publish and Subscribe
' This sample publishes and subscribes to a specified topic.
' Text you enter is published and then received by all subscribers.
```

```
Dim newProgressJMS As New SonicMQ.SonicMQ
```

```
' The WithEvents keyword is required to receive asynchronous events
Dim WithEvents ProgressJMS As SonicMQ.SonicMQ
Attribute ProgressJMS.VB_VarHelplID = -1
```

```
Dim sessionid As Long
Dim topicid As Long
Dim subscriberid As Long
Dim publisherid As Long
Dim messageid As Long
Dim mapmsgid As Long
Dim pubresult As Long
Dim setresult As Long
Dim result As Long
Dim username As String
Dim password As String
Dim connectresult As String
```

```
Private Sub Form_Load()
    Set ProgressJMS = newProgressJMS
    Dim listener As Long

    ' Set up parameters to connect to a broker running on the
    ' same machine. Modify username and password if the broker
    ' is security enabled.

    username = "ActiveXUser"
    password = "password"

    ProgressJMS.setBrokerURL ("localhost:2506")
    ProgressJMS.setClientID ("ActiveXClient")
    ProgressJMS.setUsername (username)
    ProgressJMS.setPassword (password)

    ' Connect to broker.
    connectresult =
        ProgressJMS.jms_CreateTopicConnection_withDefaultUser()
    If (connectresult < 0) Then
        failmsg = "jons_CreateTopicConnection_withDefaultUser
        failed"
        GoTo ErrorHandler
    End If
```

```

' Create a Pub/Sub session
  sessi on i d = ProgressJMS.j ms_Cre ateTopi cSessi on
                (Fal se, ProgressJMS.j ms_Sessi on_AUTO_ACKNOWLEDGE)
  If (sessi on i d < 0) Then
    fai lmsg = "j ms_Cre ateTopi cSessi on fai led"
    GoTo ErrorHandler
  End If

' Identify the topic that we will publish and subscribe to
  topi c i d = ProgressJMS.j ms_Cre ateTopi c
              (sessi on i d, "j ms. sampl es. chat")
  If (topi c i d < 0) Then
    fai lmsg = "j ms_Cre ateTopi c fai led"
    GoTo ErrorHandler
  End If

' Subscribe to the topic
  subscri ber i d = ProgressJMS.j ms_Cre ateSubscri ber
                  (sessi on i d, topi c i d)
  If (subscri ber i d < 0) Then
    fai lmsg = "j ms_Cre ateSubscri ber fai led"
    GoTo ErrorHandler
  End If

' Create a publisher to the topic
  publi sher i d = ProgressJMS.j ms_Cre atePubli sher
                  (sessi on i d, topi c i d)
  If (publi sher i d < 0) Then
    fai lmsg = "j ms_Cre atePubli sher fai led"
    GoTo ErrorHandler
  End If

' We will listen for messages asynchronously, create a listener
  li stener = ProgressJMS.j ms_Cre ateMessageLi stener()
  If (li stener < 0) Then
    fai lmsg = "j ms_Cre ateMessageLi stener fai led"
    GoTo ErrorHandler
  End If

' Attach the listener to the subscription created above
  resul t = ProgressJMS.j ms_MessageConsumer_setMessageLi stener
           (subscri ber i d, li stener)
  If (resul t < 0) Then
    fai lmsg = "j ms_MessageConsumer_setMessageLi stener fai led"
    GoTo ErrorHandler
  End If

' Now that setup is complete, start the Connection
  resul t = ProgressJMS.j ms_Connecti on_start()
  If (resul t < 0) Then
    fai lmsg = "j ms_Connecti on_start fai led"
    GoTo ErrorHandler
  End If

  Exi t Sub

```

```
ErrorHandler:
    ProgressJMS.jms_Connection_close
    MsgBox failmsg, , "SonicMQ Error"
    Exit Sub

End Sub

' Publishes the text entered in the MessageText field
Private Sub Publish_Click()
    messageid = ProgressJMS.jms_Session_createTextMessage
                (sessionid)
    If (messageid >= 0 And MessageText.Text <> "") Then
        setresult = ProgressJMS.jms_TextMessage_setText
                    (messageid, username + ": " +
MessageText.Text)
        pubresult = ProgressJMS.jms_PublishMessage
                    (publisherid, messageid)
    ' IMPORTANT - free up the message when we're done
        ProgressJMS.jms_Message_free (messageid)
    End If
End Sub

' Disconnect from the broker on form unload
Private Sub Form_Unload(Cancel As Integer)
    ProgressJMS.jms_Connection_close
End Sub

' Receives messages published to the topic, displays them in
MessageList
Private Sub ProgressJMS_onJMSMessage
    (ByVal OnJMSMessageEvent1 As Object)
    Dim msgtype As String
    Dim messageid As Integer
    messageid = OnJMSMessageEvent1.getMessageID()
    If (messageid >= 0) Then
        msgtype = ProgressJMS.jms_CheckMessageType(messageid)

    ' Check message type: this sample only works with text messages
    If (msgtype = "TextMessage") Then
        MessageList.Text = MessageList.Text +
            ProgressJMS.jms_TextMessage_getText(messageid) +
            Chr(13) +
            Chr(10)
    Else
        MessageList.Text = "Unknown message type received"
    End If

    ' IMPORTANT - free up the message when we're done
    ProgressJMS.jms_Message_free (messageid)
End Sub
End Sub
```



```
Private Sub ProgressJMS_onJMSException  
    (ByVal OnJMSExceptionEvent As Object)  
    MessageList.Text = CStr(OnJMSExceptionEvent.getJMSExceptionID()) +  
        " : " +  
        OnJMSExceptionEvent.getJMSExceptionText()  
End Sub  
  
Private Sub Clear_Click()  
    MessageList.Text = ""  
End Sub
```

Tips and Techniques for SonicMQ ActiveX/COM

The SonicMQ ActiveX/COM control optimizes the interface between the JMS objects and methods and the ActiveX/COM interface.

Identifiers

An identifier is a non-negative value returned when a function call returns successfully. Identifiers with negative values, like status codes, indicate a failure.

Identifiers are defined for the SonicMQ ActiveX/COM control so that its API flattens the object-oriented structure of the JMS API, thus creating a single API.

For example, to create a **TopicSubscriber** object in the JMS API:

1. Create a **TopicConnection** object.
2. From the **TopicConnection** methods, create a **TopicSession** object.
3. From the **TopicSession** methods, create a **TopicSubscriber** object.

Contrast that with the single object ActiveX/COM control's API:

1. Create a **TopicConnection** by calling the create connection method `jms_CreateTopicConnection()`.
2. Create the **TopicSession** object by calling the method `jms_CreateTopicSession()`. The ActiveX/COM control puts the **TopicSession** object into a table in memory and a `sessionID` value is returned to the caller.

3. Create the `TopicSubscriber` object by calling the method `jms_CreateSubscriber()` and passing the `SessionID` and `TopicID` as parameters so that the `TopicSubscriber` object can be associated with the `TopicSession` object.

Session Identifier

The session identifier methods for both domains are:

Object	Publish and Subscribe Domain	Point-to-point Domain
Producer	<code>int jms_TopicPublisher_getSessionID (int publisherID);</code>	<code>int // sessionID jms_QueueSender_getSessionID (int senderID);</code>
Consumer	<code>int jms_TopicSubscriber_getSessionID (int subscriberID);</code>	<code>int // sessionID jms_QueueReceiver_getSessionID (int receiverID);</code>
Destination	<code>int jms_Topic_getSessionID (int topicID);</code>	<code>int // sessionID jms_Queue_getSessionID (int queueID);</code>
Browser (PTP)	<i>n/a</i>	<code>int // sessionID jms_QueueBrowser_getSessionID (int qBrowserID);</code>

Looking Up the Chain of Objects

It can be useful to look up the chain of objects, for example, to identify the parent `TopicSession` object for a `TopicSubscriber`. But in the flattened API of the SonicMQ ActiveX/COM Control this is not possible. Methods are included to resolve these problems, for example:

```
jms_TopicSubscriber_getSessionID()
```

This returns the `sessionID` of the subscriber's `TopicSession` object.

Asynchronous Delivery

To support asynchronous message delivery, `JMS MessageListener` objects are included to attach to the `MessageConsumers`. To associate the `ListenerID` with a `MessageListener`, the `ListenerID` can be passed to the `jms_MessageConsumer_setMessageListener()` method. As a single `MessageListener` can attach to multiple `MessageConsumers`, the SonicMQ ActiveX/COM control lets a `jms_CreateMessageListener()` method create a `MessageListener` and return a `ListenerID` to the caller.

Handling Messages

The JMS API can use a single message handler to receive all Message types. After the Java application receives a Message object, it uses the Java instanceof operator to determine the type of the Message; for example, TextMessage. Because applications using the SonicMQ ActiveX/COM control are aware only of a messageID, the instanceof operation is not viable. A new method is added to the ActiveX/COM API to allow a message's type to be determined:

```
jms_CheckMessageType( )
```

where the returned value will be one of the following String values:

"XMLMessage"	"BytesMessage"
"TextMessage"	"ObjectMessage"
"MapMessage"	"StreamMessage"
"Message"	"Unknown message type"

XML Messages

SonicMQ has added an XMLMessage type to its supported Message object types. However, only the method's `jms_XMLMessage_getText()` and `jms_XMLMessage_setText()` are applicable under ActiveX/COM, allowing the message's XML character data to be accessed. Methods to manipulate the DOM object are not supported under ActiveX/COM in this release.

Resource Management

In ActiveX/COM applications that interface with SonicMQ, you should free memory resources allocated to the message as soon as you are sure that the message is no longer of interest:

```
jms_Message_free( int messageID );
```

See [“Visual Basic Code for the ActiveX/COM Sample”](#) for an example of how freeing resources is used by both the publisher and the subscriber of the message.

Events

Two event methods are available in the SonicMQ ActiveX/COM control, the `onJMSMessage` event and the `onException` event.

Asynchronous OnJMSMessage Event

The SonicMQ ActiveX/COM control produces an `OnJMSMessageEvent` whenever the control has asynchronously received a message for the application.

The callback method or Event Sink defined in an application for `OnJMSMessageEvents` is the `onJMSMessage()` method that accepts the event as a parameter.

The `OnJMSMessageEvent` requires that you use the `getMessageID()` method to return the `messageID` of the asynchronous message that is being delivered.

On Exception Events

The SonicMQ ActiveX/COM control produces a `OnJMSExceptionEvent` that is triggered whenever a Java `Exception` prevents a method from completing successfully.

As most methods typically return a negative error code in the event of failure, implementing support for `OnJMSException` events in an application is not strictly required.

The callback method or Event Sink defined in an application for `OnJMSExceptionEvents` needs an `onJMSException()` method, which accepts that event as a parameter.

The `OnJMSExceptionEvent` uses the `getJMSExceptionText()` method among others to retrieve the text of the Java `Exception`.

Additional Exception methods are:

- `String`
`getJMSGeneralException()`;
- `int`
`getJMSGeneralExceptionCode()`;

Connections

An instance of the SonicMQ ActiveX/COM control supports a single connection to the message server. Once a connection is established, any subsequent calls to create a connection are ignored. As there are cases when multiple connections are needed, you can use multiple instances of the SonicMQ ActiveX/COM control to establish the required number of connections.

True ActiveX/COM Properties

True ActiveX/COM properties are properties where:

- The properties can be found through introspection of the control by the container.
- A get method and set method are provided for the property.

The SonicMQ ActiveX/COM control makes the connection parameters available as true ActiveX/COM properties as shown in with their respective get and set methods. The connection parameters of the true ActiveX/COM properties in the SonicMQ ActiveX/COM control are shown in [Table 24](#).

Table 24. True ActiveX/COM Properties in the SonicMQ ActiveX/COM Control

<i>Connection Parameter</i>	<i>set Method</i>	<i>get Method</i>
UserName	setUsername(String user);	String getUsername();
Password	setPassword(String password);	String getPassword();
ClientID	setClientID(String clientID);	String getClientID();
URL	setBrokerURL(String brokerURL);	String getBrokerURL();

Return Values

When the SonicMQ ActiveX/COM control's public methods do not return data from the fields of a message, they return an integer status code or an identifier.

Status Codes

A status code of **0** is returned when an operation succeeds and a negative value is returned when an operation fails. The specific negative value has meaning, indicative of the general area that caused the failure.

For example, to create a `TopicSubscriber` object in the JMS API, you do the following:

1. Create a `TopicConnection` object.
2. From the `TopicConnection` methods, create a `TopicSession` object.
3. From the `TopicSession` methods, create a `TopicSubscriber` object.

Contrast that with the single object ActiveX/COM control's API:

1. Create a `TopicConnection` by calling the create connection method `jms_CreateTopicConnection()`.
2. Create the `TopicSession` object by calling the method `jms_CreateTopicSession()`. The ActiveX/COM control puts the `TopicSession` object into a table in memory and a `sessionId` value is returned to the caller.
3. Create the `TopicSubscriber` object by calling the method `jms_CreateSubscriber()`, passing the `sessionId` and `topicId` as parameters so that the `TopicSubscriber` object can be associated with the `TopicSession` object.

Enumerations

Because Java enumerations cannot be passed to a non-Java application, `javax.jms` methods that return an enumeration are handled in the ActiveX/COM control by four methods that comprise a simple loop that replicates enumeration:

1. Point to the next element.
2. Get that element.
3. Determine whether there are more elements.
4. If there are more elements, go to Step 1.

Constants

While the JMS API includes several static variables, the ActiveX/COM Control does not allow calling applications to use these static variables. This limitation is overcome by wrapping the constants into methods.

For example, the current JMS Specification defines three static variables that represent the **Acknowledgement Modes** for a **Session**:

- `static final int AUTO_ACKNOWLEDGE = 1;`
- `static final int CLIENT_ACKNOWLEDGE = 2;`
- `static final int DUPS_OK_ACKNOWLEDGE = 3;`

In the SonicMQ ActiveX/COM control's API, the **Acknowledgement Modes** for a **Session** are obtained by calling corresponding methods:

- `int jms_Session_AUTO_ACKNOWLEDGE();`
- `int jms_Session_CLIENT_ACKNOWLEDGE();`
- `int jms_Session_DUPS_OK_ACKNOWLEDGE();`

Syntax for SonicMQ ActiveX/COM Method Names

The standard naming method for SonicMQ API names in the SonicMQ ActiveX/COM control API is:

`"jms_" + JMS API Interface class name + "_" + method name`

For example, the `start()` method in the `Connection` interface class is:

```
jms_Connection_start( )
```

Duplicate Names Are Differentiated

When duplicates occur, a distinguishing name is added:

`"jms_" + JMS API Interface class name + "_" + method name + "_" + some distinguishing name`

Duplicates occur primarily because overloading a method name. The two most common cases are the constants as discussed above and the handling of Java's ability to manage overloaded constructors that needs to be discretely stated in ActiveX/COM implementations.

Java Method Overloading Is Handled

As ActiveX/COM does not allow two methods to have the same name where Java does support method overloading, enabling different sets of arguments to perform variations of the basic function. To accommodate the JMS functionality, the ActiveX/COM control names extend names to differentiate the variants. For example, the `MessageConsumer` methods:

- `Message receive()`
- `Message receive(long timeout)`

are presented in the SonicMQ ActiveX/COM control's API as:

- `int // messageID
jms_MessageConsumer_receive(int consumerID);`
- `int // messageID
jms_MessageConsumer_receive_withTimeout(int consumerID, long
timeout);`

Interface Class Names Are Often Omitted

In most cases, the complete interface name is part of a method name, for example, `jms_MessageConsumer_receive()`

Some common names are simplified to keep names brief. For example:

- `jms_CreateTopicSession()` does not need to be qualified by including `TopicConnection` as part of its name.
- `jms_Publish()` can only belong to the `TopicPublisher`, so that Interface's class name was dropped from method's name.

Interface Mappings

Table 25 presents the set of tables in this chapter that detail the mapping of javax.jms constructors and methods to the commands used with the ActiveX/COM control.

Table 25. Interface Mapping from SonicMQ to the ActiveX/COM Control

Interface
Table 26, “Connection Interface” on page 242.
Table 27, “Session Interface” on page 243.
Table 28, “MessageConsumer Interface” on page 244.
Table 29, “MessageListener Interface” on page 245.
Table 30, “MessageProducer Interface” on page 245.
Table 31, “DeliveryMode Interface” on page 246.
Table 32, “TopicConnectionFactory Interface” on page 246.
Table 33, “TopicConnection Interface” on page 247.
Table 34, “TopicSession Interface” on page 247.
Table 35, “Topic Interface (Extends Destination)” on page 248.
Table 36, “TopicPublisher Interface” on page 248.
Table 37, “TopicRequestor and TemporaryTopic (Extends Topic) Interfaces” on page 248.
Table 38, “TopicSubscriber Interface” on page 249.
Table 39, “QueueConnectionFactory Interface” on page 249.
Table 40, “QueueConnection Interface” on page 249.
Table 41, “QueueSession Interface” on page 250.
Table 42, “Queue Interface (Extends Destination)” on page 250.
Table 43, “QueueSender Interface (Extends MessageProducer)” on page 251.

Table 25. Interface Mapping from SonicMQ to the ActiveX/COM Control

<i>Interface</i>
Table 44, “QueueRequestor and TemporaryQueue (Extends Queue) Interfaces” on page 251.
Table 45, “QueueReceiver Interface (Extends MessageConsumer)” on page 251.
Table 46, “QueueBrowser Interface” on page 252.
Table 47, “Message Interface” on page 252.
Table 48, “BytesMessage Interface (Extends Message)” on page 256.
Table 49, “MapMessage Interface (Extends Message)” on page 258.
Table 50, “StreamMessage Interface (Extends Message)” on page 260.
Table 51, “TextMessage Interface (Extends Message)” on page 261.
Table 52, “XMLMessage Interface (Extends TextMessage)” on page 262.
Table 53, “Other Interfaces” on page 262.

Connections and Sessions

Table 26. Connection Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
String getClientID()	String getClientID();
void setClientID(String clientID)	void setClientID(String clientID);
void setExceptionHandler(ExceptionListener listener)	-
void start()	int // status code jms_Connecti_on_start()
void stop()	int // status code jms_Connecti_on_stop()

Table 26. Connection Interface (*continued*)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
void close()	int // status code jms_Connecti on_cl ose()

Table 27. Session Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
static final int AUTO_ACKNOWLEDGE = 1;	int jms_Sessi on_AUTO_ACKNOWLEDGE ();
static final int CLIENT_ACKNOWLEDGE = 2;	int jms_Sessi on_CLI ENT_ACKNOWLEDGE ();
static final int DUPS_OK_ACKNOWLEDGE = 3;	int jms_Sessi on_DUPS_OK_ACKNOWLEDGE();
createMessage()	int // messageID jms_Sessi on_createMessage(int sessi onID);
createBytesMessage()	int // messageID jms_Sessi on_createBytesMessage (int sessi onID);
createMapMessage()	int // messageID jms_Sessi on_createMapMessage (int sessi onID);
createStreamMessage()	int // messageID jms_Sessi on_createStreamMessage (int sessi onID);
createTextMessage()	int // messageID jms_Sessi on_createTextMessage (int sessi onID);
createTextMessage(String string)	int // messageID jms_Sessi on_createTextMessage_wi thBody (int sessi onID, String body);
boolean getTransacted()	Boolean jms_Sessi on_getTransacted(int sessi onID);
void commit()	int // status code jms_Sessi on_commi t(int sessi onID);
void rollback()	int // status code jms_Sessi on_rol l back(int sessi onID);

Table 27. Session Interface (*continued*)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
void close()	int // status code jms_Session_close(int sessionID);
void recover()	int // status code jms_Session_recover(int sessionID);
getMessageListener()	-
void setMessageListener(MessageListener listener)	-
onException(JMSException exception);	-

Producers and Consumers

Table 28. MessageConsumer Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
String getMessageSelector()	String jms_MessageConsumer_getMessageSelector (int consumerID);
receive()	int // messageID jms_MessageConsumer_receive (int consumerID);
receive(long timeout)	int // messageID jms_MessageConsumer_receive_withTimeout (int consumerID, long timeout);
Message receiveNoWait()	int // messageID jms_MessageConsumer_receiveNoWait (int consumerID);
void close()	int // status code jms_MessageConsumer_close (int consumerID);

Table 29. MessageListener Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
<code>void onMessage(Message message);</code>	<code>jms_CreateMessageListener</code>
	<code>int // listenerID jms_MessageConsumer_getMessageListener (int consumerID)</code>
	<code>int // status code jms_MessageConsumer_setMessageListener (int consumerID, int listenerID)</code> where <code>listenerID</code> is generated by the <code>jms_CreateMessageListener</code> method

Table 30. MessageProducer Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
<code>setDisableMessageID(boolean value)</code>	<code>int // status code jms_MessageProducer_setDisableMessageID (int producerID, boolean value);</code>
<code>boolean getDisableMessageID()</code>	<code>Boolean jms_MessageProducer_getDisableMessageID (int producerID);</code>
<code>setDisableMessageTimestamp(boolean value)</code>	<code>int // status code jms_MessageProducer_setDisableMessageTimestamp (int producerID, boolean value);</code>
<code>boolean getDisableMessageTimestamp()</code>	<code>Boolean jms_MessageProducer_getDisableMessageTimestamp (int producerID);</code>
<code>setDeliveryMode(int deliveryMode)</code>	<code>int // status code jms_MessageProducer_setDeliveryMode (int producerID, int deliveryMode);</code>
<code>int getDeliveryMode()</code>	<code>int // integer value (otherwise null) jms_MessageProducer_getDeliveryMode (int producerID);</code>
<code>setPriority(int priority)</code>	<code>int // status code jms_MessageProducer_setPriority (int producerID, int priority);</code>

Table 30. MessageProducer Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
int getPriority()	int // integer value (otherwise null) jms_MessageProducer_getPriority (int producerID);
setTimeToLive(long timeToLive)	int // status code jms_MessageProducer_setTimeToLive (int producerID, long timeToLive);
int getTimeToLive()	int // integer value (otherwise null) jms_MessageProducer_getTimeToLive (int producerID);
close()	int // status code jms_MessageProducer_close(int producerID);

Table 31. DeliveryMode Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
static final int NON_PERSISTENT = 1;	int jms_DeliveryMode_NON_PERSISTENT();
static final int PERSISTENT = 2;	int jms_DeliveryMode_PERSISTENT();

Publish and Subscribe (Topics)

Table 32. TopicConnectionFactory Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
createTopicConnection()	int // status code jms_CreateTopicConnection_withDefaultUser()
createTopicConnection (String userName, String password)	int // status code jms_CreateTopicConnection (String userName, String password)

Table 33. TopicConnection Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
<code>createTopicSession</code> (boolean transacted, int acknowledgeMode)	int // sessionId <code>jms_CreateTopicSession</code> (boolean transacted, int acknowledgeMode)

Table 34. TopicSession Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
<code>createTopic</code> (String topicName)	int // topicID <code>jms_CreateTopic</code> (int sessionId, String topicName);
<code>createSubscriber</code> (Topic topic)	int // subscriberID <code>jms_CreateSubscriber</code> (int sessionId, int topicID);
<code>createSubscriber</code> (Topic topic, String messageSelector, boolean noLocal)	int // subscriberID <code>jms_CreateSubscriber_withSelector</code> (int sessionId, int topicID, String messageSelector, Boolean noLocal);
<code>createDurableSubscriber</code> (Topic topic, String name)	int // subscriberID <code>jms_CreateDurableSubscriber</code> (int sessionId, int topicID, String name);
<code>createDurableSubscriber</code> (Topic topic, String name, String messageSelector, boolean noLocal)	int // subscriberID <code>jms_CreateDurableSubscriber_withSelector</code> (int sessionId, int topicID, String name, String messageSelector, Boolean noLocal);
<code>createPublisher</code> (Topic topic)	int // publisherID <code>jms_CreatePublisher</code> (int sessionId, int topicID);
<code>createTemporaryTopic</code> ()	int // topicID <code>jms_CreateTemporaryTopic</code> (int SessionID);
<code>unsubscribe</code> (String name)	int // status code <code>jms_Unsubscribe</code> (int sessionId, String topicName);

Table 35. Topic Interface (Extends Destination)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
String getTopicName()	String jms_Topic_getTopicName(int topicID);
String toString();	String jms_Topic_toString(int topicID);

Table 36. TopicPublisher Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
getTopic()	int // topicID jms_TopicPublisher_getTopic(int publisherID);
publish(Message message)	int // status code jms_PublishMessage (int publisherID, int messageID);
publish(Message message, int deliveryMode, int priority, long timeToLive)	int // status code jms_PublishMessage_wi thCondi ti ons (int publisherID, int messageID, int deliveryMode, int priority, long timeToLive);
publish(Topic topic, Message message)	int // status code jmsPublishMessageToTopic(publisherID, int topicID, int messageID);
publish(Topic topic, Message message, int deliveryMode, int priority, long timeToLive)	int // status code jmsPublishMessageToTopic_wi thCondi ti ons(publisherID, int topicID, int messageID, int deliveryMode, int priority, long timeToLive);

Table 37. TopicRequestor and TemporaryTopic (Extends Topic) Interfaces

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
TopicRequestor (TopicSession session, Topic topic)	-
request(Message message)	-

Table 37. TopicRequestor and TemporaryTopic (Extends Topic) Interfaces

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
close()	-
Temporary Topic: delete()	int // status code jms_TemporaryTopic_delete(int topicID);

Table 38. TopicSubscriber Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
getTopic()	int // topicID jms_TopicSubscriber_getTopic(int subscriberID);
getNoLocal()	Boolean jms_TopicSubscriber_getNoLocal(int subscriberID);

Point-to-point (Queues)

Table 39. QueueConnectionFactory Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
createQueueConnection()	int // status code jms_CreateQueueConnection_withDefaultUser();
createQueueConnection (String userName, String password)	int // status code jms_CreateQueueConnection (String userName, String password);

Table 40. QueueConnection Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
createQueueSession (boolean transacted, int acknowledgeMode)	int // sessionID jms_createQueueSession (boolean transacted, int acknowledgeMode);

Table 41. QueueSession Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
createQueue(String queueName)	int // queueID jms_CreateQueue(int sessionID, String name);
createReceiver(Queue queue)	int // qReceiverID jms_CreateQueueReceiver(int sessionID, int queueID);
createReceiver(Queue queue, String messageSelector)	int // qReceiverID jms_CreateQueueReceiver_withSelector (int sessionID, int queueID, String messageSelector);
createSender(Queue queue)	int // qSenderID jms_CreateQueueSender(int sessionID, int queueID);
createBrowser(Queue queue)	int // qBrowserID jms_CreateQueueBrowser (int sessionID, int queueID);
createBrowser(Queue queue, String messageSelector)	int // qBrowserID jms_CreateQueueBrowser_withSelector (int sessionID, int queueID, String messageSelector);
createTemporaryQueue()	int // queue jms_CreateTemporaryQueue(int sessionID);

Table 42. Queue Interface (Extends Destination)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
getQueueName()	String jms_Queue_getQueueName(int queueID);
String toString();	String jms_Queue_toString(int queueID);

Table 43. QueueSender Interface (Extends MessageProducer)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
getQueue()	int // queueID jms_QueueSender_getQueue(int qSenderID);
send(Message message)	int // status code jms_SendMessage (int qSenderID, int messageID);
send(Message message, int deliveryMode, int priority, long timeToLive)	int // status code jms_SendMessage_wi thCondi ti ons (int qSenderID, int messageID, int deliveryMode, int priority, long timeToLi ve);
send(Queue queue, Message message)	int // status code jms_SendMessageToQueue (int qSenderID, int queueID, int messageID);
send(Queue queue, Message message, int deliveryMode, int priority, long timeToLive)	int // status code jms_SendMessageToQueue_wi thCondi ti ons (int qSenderID, int queueID, int messageID, int deliveryMode, int priority, long timeToLi ve);

Table 44. QueueRequestor and TemporaryQueue (Extends Queue) Interfaces

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
QueueRequestor(QueueSessi on sessi on, Queue queue)	-
request(Message message)	-
close()	-
<i>TemporaryQueue:</i> delete()	int // status code jms_TemporaryQueue_del ete (int queueID);

Table 45. QueueReceiver Interface (Extends MessageConsumer)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
getQueue()	int // queueID jms_QueueRecei ver_getQueue(int qRecei verID);

Table 46. QueueBrowser Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
getQueue()	int // queueID jms_QueueBrowser_getQueue(int qBroswerID)
String getMessageSelector())	String jms_QueueBrowser_getMessageSelector(int qBrowserID);
Enumeration getEnumeration()	int // Message Queue Enumeration ID jms_QueueBrowser_getEnumeration(int qBrowserID);
	int // Message ID jms_QueueBrowserEnumeration_nextElement(int enumerationID) ;
	Boolean // Message ID jms_QueueBrowserEnumeration_hasMoreElements(int enumerationID)
close()	int // status code jms_QueueBrowser_close(int qBrowserID);

Messages

Table 47. Message Interface

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
static final int DEFAULT_DELIVERY_MODE = -1;	int jms_Message_DEFAULT_DELIVERY_MODE();
static final int DEFAULT_PRIORITY = -1;	int jms_Message_DEFAULT_PRIORITY();
static final int DEFAULT_TIME_TO_LIVE = -1;	int jms_Message_DEFAULT_TIME_TO_LIVE();
String getJMSMessageID()	String jms_Message_getJMSMessageID(int messageID);
void setJMSMessageID(String id)	int // status code jms_Message_setJMSMessageID (int messageID, String id);
long getJMSTimestamp()	Long jms_Message_getJMSTimestamp(int messageID);

Table 47. Message Interface (continued)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
setJMSTimestamp(long timestamp)	int // status code jms_Message_setJMSTimestamp (int messageID, long timestamp);
byte [] getJMSCorrelationIDsAsBytes()	byte[] // jms_Message_getJMSCorrelationIDsAsBytes (int messageID);
byte[] setJMSCorrelationIDsAsBytes (byte[] correlationID)	int // status code jms_Message_setJMSCorrelationIDsAsBytes (int messageID, byte[] correlationID);
String setJMSCorrelationID (String correlationID)	int // status code jms_Message_setJMSCorrelationID (int messageID, String correlationID);
getJMSCorrelationID()	String jms_Message_getJMSCorrelationID(int messageID);
getJMSReplyTo()	int // destinationID jms_Message_getJMSReplyTo(int messageID);
setJMSReplyTo (Destination replyTo)	int // status code jms_Message_setJMSReplyTo(int messageID, int destinationID);
int getJMSDestination()	int // destinationID jms_Message_getJMSDestination(int messageID);
setJMSDestination (Destination destination)	int // status code jms_Message_setJMSDestination (int messageID, int destinationID);
int getJMSDeliveryMode()	int jms_Message_getJMSDeliveryMode(int messageID);
setJMSDeliveryMode (int deliveryMode)	int // status code jms_Message_setJMSDeliveryMode (int messageID, int deliveryMode);
boolean getJMSRedelivered()	Boolean jms_Message_getJMSRedelivered(int messageID);
setJMSRedelivered (boolean redelivered)	int // status code jms_Message_setJMSRedelivered(int messageID);
String getJMSType()	String jms_Message_getJMSType(int messageID);

Table 47. Message Interface (continued)

javax.jms API	SonicMQ ActiveX/COM API
setJMSType(String type)	int // status code jms_Message_setJMSType (int messageID , String type);
long getJMSExpiration()	long jms_Message_getJMSExpiration (int messageID);
setJMSExpiration(long expiration)	int // status code jms_Message_setJMSExpiration (int messageID, long expiration);
int getJMSPriority()	int jms_Message_getPriority (int messageID);
setJMSPriority(int priority)	int // status code jms_Message_setJMSPriority (int messageID, int priority);
clearProperties()	int // status code jms_Message_clearProperties (int messageID);
boolean propertyExists(String name)	Boolean jms_Message_propertyExists (int messageID String name);
boolean getBooleanProperty(String name)	Boolean jms_Message_getBooleanProperty (int messageID, String name);
byte getByteProperty(String name)	Byte jms_Message_getByteProperty (int messageID, String name);
short getShortProperty(String name)	Short jms_Message_getShortProperty (int messageID, String name);
int getIntProperty(String name)	Integer jms_Message_getIntProperty (int messageID, String name);
long getLongProperty(String name)	Long jms_Message_getLongProperty (int messageID, String name);
float getFloatProperty(String name)	Float jms_Message_getFloatProperty (int messageID, String name);

Table 47. Message Interface (continued)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
double getDoubleProperty(String name)	Double jms_Message_getDoubleProperty (int messageID, String name);
String getStringProperty(String name)	String jms_Message_getStringProperty (int messageID, String name);
Object getObjectProperty(String name)	Not implemented.
Enumeration getPropertyNames()	String[] jms_Message_getPropertyNames(int messageID);
setBooleanProperty (String name, boolean value)	int // status code jms_Message_setBooleanProperty (int messageID, String name, Boolean value);
setByteProperty (String name, byte value)	int // status code jms_Message_setByteProperty (int messageID, String name, Byte value);
setShortProperty (String name, short value)	int // status code jms_Message_setShortProperty (int messageID, String name, Short value);
setIntProperty (String name, int value)	int // status code jms_Message_setIntProperty (int messageID, String name, Integer value);
setLongProperty (String name, long value)	int // status code jms_Message_setLongProperty (int messageID, String name, Long value);
setFloatProperty (String name, float value)	int // status code jms_Message_setFloatProperty (int messageID, String name, Float value);
setDoubleProperty (String name, double value)	int // status code jms_Message_setDoubleProperty (int messageID, String name, Double value);
setProperty (String name, String value)	int // status code jms_Message_setStringProperty (int messageID, String name, String value);
setProperty (String name, Object value)	int // status code jms_Message_setObjectProperty (int messageID, String name, Object value);

Table 47. Message Interface (continued)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
acknowledge()	int // status code jms_Message_acknowledge(int messageID);
clearBody()	int // status code jms_Message_clearBody(int messageID);

Table 48. BytesMessage Interface (Extends Message)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
boolean readBoolean()	Boolean jms_BytesMessage_readBoolean(int messageID);
byte readByte()	Byte jms_BytesMessage_readByte(int messageID);
int readUnsignedByte()	Integer jms_BytesMessage_readUnsignedByte(int messageID);
short readShort()	Short jms_BytesMessage_readShort(int messageID);
int readUnsignedShort()	Integer jms_BytesMessage_readUnsignedShort(int messageID);
char readChar()	char jms_BytesMessage_readChar(int messageID);
int readInt()	Integer jms_BytesMessage_readInt(int messageID);
long readLong()	Long jms_BytesMessage_readLong(int messageID);
float readFloat()	Float jms_BytesMessage_readFloat(int messageID);
double readDouble()	Double jms_BytesMessage_readDouble(int messageID);
String readUTF()	String jms_BytesMessage_readUTF(int messageID);

Table 48. BytesMessage Interface (Extends Message) (continued)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
int readBytes (byte[] value)	Byte[] jms_BytesMessage_readBytes (int messageID, int length);
int readBytes (byte[] value, int length)	Not implemented.
writeln(boolean value)	int // status code jms_BytesMessage_writeln (int messageID, boolean value);
writeln(byte value)	int // status code jms_BytesMessage_writelnByte (int messageID, byte value);
writeln(short value)	int // status code jms_BytesMessage_writelnShort (int messageID, short value);
writeln(char value)	int // status code jms_BytesMessage_writelnChar (int messageID, char value);
writeln(int value)	int // status code jms_BytesMessage_writelnInt (int messageID, int value);
writeln(long value)	int // status code jms_BytesMessage_writelnLong (int messageID, long value);
writeln(float value)	int // status code jms_BytesMessage_writelnFloat (int messageID, float value);
writeln(double value)	int // status code jms_BytesMessage_writelnDouble (int messageID, double value);
writelnUTF(String value)	int // status code jms_BytesMessage_writelnUTF (int messageID, String value);
writelnBytes(byte[] value)	int // status code jms_BytesMessage_writelnBytes (int messageID, byte[] value);
writelnBytes (byte[] value, int offset, int length)	int // status code jms_BytesMessage_writelnBytes_atOffset (int messageID, byte[] value, int offset, int length);

Table 48. BytesMessage Interface (Extends Message) (continued)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
writeObject(Object value)	int // status code jms_BytesMessage_writeObject(int messageId, Object value);
reset()	int // status code jms_BytesMessage_reset(int messageId);

Table 49. MapMessage Interface (Extends Message)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
boolean getBoolean(String name)	Boolean jms_MapMessage_getBoolean(int messageId, String name);
byte getBytes(String name)	Byte jms_MapMessage_getByte(int messageId, String name);
char getChar(String name)	char jms_MapMessage_getChar(int messageId, String name);
int getInt(String name)	int // integer value jms_MapMessage_getInt(int messageId, String name);
long getLong(String name)	long jms_MapMessage_getLong(int messageId, String name);
float getFloat(String name)	float jms_MapMessage_getFloat(int messageId, String name);
double getDouble(String name)	double jms_MapMessage_getDouble(int messageId, String name);
String getString(String name)	String jms_MapMessage_getString(int messageId, String name);
byte[] getBytes(String name)	byte[] jms_MapMessage_getBytes(int messageId, String name);
Object getObject(String name)	Not implemented.
Enumeration getMapNames()	String[] jms_MapMessage_getMapNames(int messageId);
setBoolean (String name, boolean value)	int // status code jms_MapMessage_setBoolean(int messageId, String name, boolean value);

Table 49. MapMessage Interface (Extends Message) (continued)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
setByte (String name, byte value)	int // status code jms_MapMessage_setByte (int messageID, String name, byte value);
setShort (String name, short value)	int // status code jms_MapMessage_setShort (int messageID, String name, short value);
setChar (String name, char value)	int // status code jms_MapMessage_setChar (int messageID, String name, char value);
setInt (String name, int value)	int // status code jms_MapMessage_setInt (int messageID, String name, int value);
setLong (String name, long value)	int // status code jms_MapMessage_setLong (int messageID, String name, long value);
setFloat (String name, float value)	int // status code jms_MapMessage_setFloat (int messageID, String name, float value);
setDouble (String name, double value)	int // status code jms_MapMessage_setDouble (int messageID, String name, double value);
setString (String name, String value)	int // status code jms_MapMessage_setString (int messageID, String name, String value);
setBytes (String name, byte[] value)	int // status code jms_MapMessage_setBytes (int messageID, String name, byte[] value);
setBytes (String name, byte[] value, int offset, int length)	int // status code jms_MapMessage_setBytes_atOffset (int messageID, String name, byte[] value , int offset, int length)
setObject (String name, Object value)	int // status code jms_MapMessage_setObject (int messageID, String name, Object value);
boolean itemExists (String name)	Boolean jms_MapMessage_itemExists (String name);

Table 50. StreamMessage Interface (Extends Message)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
boolean readBoolean()	Boolean jms_StreamMessage_readBoolean(int messageID);
byte readByte()	Byte jms_StreamMessage_readByte(int messageID);
short readShort()	Short jms_StreamMessage_readShort(int messageID);
char readChar()	char jms_StreamMessage_readChar(int messageID);
int readInt()	Integer // integer value jms_StreamMessage_readInt(int messageID);
long readLong()	Long jms_StreamMessage_readLong(int messageID);
float readFloat()	Float jms_StreamMessage_readFloat(int messageID);
double readDouble()	Double jms_StreamMessage_readDouble(int messageID);
String readString()	String jms_StreamMessage_readString(int messageID);
int readBytes(byte[] value)	byte[] jms_StreamMessage_readBytes(int messageID, int length);
Object readObject()	Not implemented.
writeBoolean (boolean value)	int // status code jms_StreamMessage_writeBoolean (int messageID, boolean value);
writeByte(byte value)	int // status code jms_StreamMessage_writeByte (int messageID, byte value);
writeShort(short value)	int // status code jms_StreamMessage_writeShort (int messageID, short value);
writeChar(char value)	int // status code jms_StreamMessage_writeChar(int messageID, char value);
writeInt(int value)	int // status code jms_StreamMessage_writeInt (int messageID, int value);

Table 50. StreamMessage Interface (Extends Message) (continued)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
wri teLong(long value)	int // status code jms_StreamMessage_wri teLong (int messageID, long value);
wri teFloat(float value)	int // status code jms_StreamMessage_wri teFloat (int messageID, float value);
wri teDouble(double value)	int // status code jms_StreamMessage_wri teDouble (int messageID, double value);
wri teString(String value)	int // status code jms_StreamMessage_wri teString (int messageID, String value);
wri teBytes(byte[] value)	int // status code jms_StreamMessage_wri teBytes (int messageID, byte[] value);
wri teBytes (byte[] value, int offset, int length)	int // status code jms_StreamMessage_wri teBytes_atOffset (int messageID, byte[] value, int offset, int length);
wri teObject(Object value)	int // status code jms_StreamMessage_wri teObject(int messageID, Object value);
reset()	int // status code jms_StreamMessage_reset(int messageID);

Table 51. TextMessage Interface (Extends Message)

<i>javax.jms API</i>	<i>SonicMQ ActiveX/COM API</i>
setText(String string)	int // status code jms_TextMessage_setText(int messageID, String textBody);
String getText()	String jms_TextMessage_getText(int messageID);

Note The XMLMessage interface extends javax.jms, referencing it as a progress.message.jclient API.

Table 52. XMLMessage Interface (Extends TextMessage)

<i>progress.message.jclient. API</i>	<i>SonicMQ ActiveX/COM API</i>
(progress.message.jclient.Session) session).createXMLMessage()	int // status code jms_Session_CreateXMLMessage(int sessionID);
	int // status code jms_Session_CreateXMLMessage_withBody (int sessionID, String textBody);

Special Purpose

The ActiveX/COM commands listed in [Table 53](#) have no similar command in either javax.jms or progress.message.jclient.

Table 53. Other Interfaces

<i>NO ANALOGOUS JMS METHOD</i>	<i>SonicMQ ActiveX/COM API</i>
SESSION ID (Topic Pub/Sub)	int jms_TopicPublisher_getSessionID(int publisherID);
	int jms_TopicSubscriber_getSessionID(int subscriberID);
	int jms_Topic_getSessionID(int topicID);
SESSION ID (Queue PTP)	int // sessionID jms_Queue_getSessionID(int queueID);
	int // sessionID jms_QueueBrowser_getSessionID(int qBrowserID);
	int // sessionID jms_QueueReceiver_getSessionID(int receiverID);
	int // sessionID jms_QueueSender_getSessionID(int senderID);

Table 53. Other Interfaces

<i>NO ANALOGOUS JMS METHOD</i>	SonicMQ ActiveX/COM API
<p>MESSAGE TYPE</p> <p>Java clients check the Message using <code>instanceOf</code> to determine the message type.</p>	<pre>String jms_CheckMessageType(int messageID);</pre>
<p>EXCEPTIONS</p>	<pre>String getJMSGeneralExcepti on();</pre> <pre>int getJMSGeneral Excepti onCode();</pre>
<p>RESOURCE MANAGEMENT</p> <p>Cues the ActiveX/COM control to free any memory that it has reserved for the indicated message.</p>	<pre>int jms_Message_free(int messageID);</pre>

About Administered Objects

The **administered objects** are objects that are defined independently of a SonicMQ message server. These objects, set in the context of an application, provide the application with deployment details by just choosing a configuration name. As a result, developers are removed from the burden of defining and maintaining configuration details.

Within the JMS specification, the objects that can be administered are:

- **Connecti onFactori es**
 - **QueueConnecti onFactory**
 - **Topi cConnecti onFactory**
- **Desti nati ons**
 - **Queue**
 - **Topi c**

These objects depend on the configuration of the message server—such as its location, its default port, as well as what other applications are running on the message server.

SonicMQ supports administered objects that are created using the SonicMQ Explorer. See the *SonicMQ Installation and Administration Guide* for information about administered objects and the SonicMQ Explorer.

Note The sample code segments in this chapter use `TopicConnections` and `Topics`. The coding for `QueueConnections` and `Queues` is similar.

Issues When Using Administered Objects

From the point of view of the programmer, there are issues that make administered objects problematic to use:

- **ConnectionFactories** —What is the right message server and its connection?
- **Destinations** — How can I avoid name conflicts?
When several applications are using similar destination naming strategies, the programmer wants assurance that inadvertent conflicts are avoided altogether.

These issues are easily managed: store administered objects in some object store and then reference the object indirectly (by name) in some context.

SonicMQ supports JNDI and a simple file store to perform these functions.

Creating New Administered Objects

You can create `ConnectionFactories` and `Destinations` as new Java objects. Typically, this is done as follows:

```
javax.jms. QueueConnectionFactory factory;  
// Create the factory as a new object. Hard code the broker name.  
factory = (new progress.message.jclient. QueueConnectionFactory  
           ("localhost:2506"));  
  
...  
// Continue, creating connection from the factory  
// Continue, creating the session from the connection.  
  
...  
// Finally, create the Queue for our application.  
javax.jms. Queue queue = session.createQueue("SampleQ1");
```

Serialized Java Objects in a File System

SonicMQ allows you to administratively store objects as Serialized Java Objects (.sjo) in a file system. By updating the .sjo objects through the SonicMQ Explorer you can isolate the programmer from specific message server configuration parameters and destination names. The task of maintaining and deploying the .sjo files remains.

Setting Up Serialized Objects

The following sample demonstrates how serialized objects can be set up. The sample assumes:

- The `TopicConnectionFactory` for the sample application is stored in the file `ChatConnectionFactory.sjo`.
- The `Topic` for the application is stored in the file `ChatTopic.sjo`.
- A new method, `readFile`, is used for both administered objects:

```

/**
 *Read an object from the given file.
 *@param filename The name of the file.
 *@return The deserialized object. If the file does not contain
 *       a valid JMS managed object or there is some
 *       read/deserialization problem, then return null.
 */
private Object readFile(String filename)
{
    try
    {
        java.io.FileInputStream fis =
            new java.io.FileInputStream(filename);

        java.io.ObjectInputStream ois =
            new java.io.ObjectInputStream(fis);

        Object readObj = ois.readObject();

        fis.close();
        return readObj;
    }
    catch(Exception e) { } // return null
    return null;
}

```

Using Serialized Objects

After setting up serialized objects, those objects can be used. Within the application code where the connection is established, use the `readFile` method to read the active `javax.jms` objects:

```
javax.jms.TopicConnectionFactory factory;
// Read in the factory from a file
factory = (javax.jms.TopicConnectionFactory)
        readFile("ChatConnectionFactory.sjo");
...
// Continue, creating connection from the factory
// Continue, creating the session from the connection.
...
// Finally, retrieve the TOPIC for our application
javax.jms.Topic topic = (javax.jms.Topic)
        readFile("ChatTopic.sjo");
```

Using JNDI to Interface With a Directory Server

Even with serialized objects in the file system, you still have to manage the deployment and naming of the `.sjo` files. The JNDI package provides better administration and standard mechanisms for deployment and naming. JNDI provides interfaces to standard directory servers such as those that are compliant with the Lightweight Directory Access Protocol (LDAP).

Important JNDI services and LDAP directory servers are distinct products that you must install and configure separate from SonicMQ. The Javasoft JNDI Web site can point you to evaluation editions of LDAP directory servers so that you can explore these services.

The way JNDI works is common to all JNDI providers; the difference is in the way you establish the naming context. This JNDI sample assumes that:

- The `QueueConnectionFactory` and `Queue` are stored in a default context.
- The name of the `QueueConnectionFactory` object is `TalkConnFactory`.
- The name for the `Queue` object is `TalkQueue`.
- The names are bound in the directory service under `acme.com`.

Warning The sample allows a possibility of raising a `javax.naming.NamingException` in the setup for the JMS application.

```

try
{
    // Set up the JNDI naming context (this example uses
    // Javasoft's LDAP SPI implementation, and requires an
    // underlying LDAP service)
    java.util.Hashtable env = new java.util.Hashtable();
    env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(javax.naming.Context.PROVIDER_URL,
            "ldap://localhost:389/o=acme.com");
    javax.naming.directory.DirContext ctx =
        new javax.naming.directory.InitialDirContext(env);
    javax.jms.QueueConnectionFactory factory;
    // Read in the factory from our naming context
    factory = (javax.jms.QueueConnectionFactory)
        ctx.lookup("cn=Tal kConnFactory");
    ...
    // Continue, creating connection from the factory
    // Continue, creating the session from the connection.
    ...
    // Finally, retrieve the Queue for our application
    javax.jms.Queue queue = (javax.jms.Queue)
        ctx.lookup("cn=Tal kQueue");
    // Close the context when done
    ctx.close();
}
catch (javax.naming.NamingException ne)
{
    ...
}

```

Index

A

- Access Control Lists 33, 36
 - report in an application 222
- acknowledgement
 - acknowledge method 100
- acknowledgeMode
 - session parameter 100
- active ping 114, 116
- ActiveX/COM
 - API mapping 241
 - check message type 263
 - creating an XML message 262
 - Exceptions 263
 - freeing resources 263
 - managing method overloading 240
 - naming conventions 239
 - overview 38
 - SessionID 262
 - syntax 239
- administered objects
 - ConnectionFactory 93
 - definition 265
 - Destinations 105
 - readFile 267
- applet 38
- application 76
- application identifier 89
- asynchronous 143, 163
 - ActiveX/COM application 234
- authentication
 - consumer 36

- in ConnectionFactory 96
 - producer 33
 - using security for the samples 42
- authorization
 - consumer 36
 - producer 33
 - using security for the samples 42
- auto acknowledgment 101

B

- body of a message
 - setting and getting 132
 - Text 85
 - XML (DOM format) 84
- browsing queues 166
 - sample 53
- Business-to-Business 175

C

- characters
 - reserved
 - in a Subscription name 196
 - in Destination names 137
 - in hierarchical name spaces 203
 - in Topic names 105, 191
 - in User names 39
 - template 203, 206

- CheckMessageType (ActiveX/COM) 263
- clearProperties 129
- client acknowledgement 101
- client identifier 90
- client session 89
- cluster 26, 175
- coerce
 - property value to permitted type 131
- commit 57
 - definition 102
- compiling modified SonicMQ samples 44, 79
- connect
 - close 107
 - events 216
 - start 107
 - stop 107
- connectID 96
- connection
 - definition 30
 - multiple 111
 - retry when broken 62
 - through ActiveX/COM control 237
- connection identifier 89
- ConnectionFactories
 - administered objects 265
 - definition 93
- constants 239
- consumer 31
- CorrelationID 124, 139
 - sample application 72
- count, prefetch 165
- createBrowser 166
- createDurableSubscriber 106, 196
- createMessage 106, 121, 193
- createPublisher 105
- createQueue 104
- createQueueConnection 99
- createQueueReceiver 106
- createQueueSession 100
- createSender 105
- createSubscriber 106, 195
- createTopic 104
- createTopicConnection 99
- createTopicSession 100

D

- dbtool 86, 88
- Dead Message Queue 175, 176
 - events 217
 - persistence 142
 - programming 168
 - QoS level 37
 - sample 63, 65
- default values 126
- delivery mode
 - default value 126
 - message header field 123
 - on the message server 142
 - producer parameter 193
- Destination
 - administered objects 265
- destination 123, 138
 - unbound 138
- disconnect
 - events 217
- Document Object Model 48, 84, 122
- documentation, available 20
- DOM 48, 84, 122, 132
- drop
 - events 216
- dropped connection 116
 - sample application 60
- duplicates_OK acknowledgement 101
- durable subscription
 - definition 196
 - handling on the message server 142
 - QoS 35
 - sample application 69
 - unsubscribing 196
- Dynamic Routing Architecture 41, 175
 - undelivered reason codes 177
 - working with a DMQ 63

E

- encryption 33
- enumeration
 - handling in ActiveX/COM 238
- enumeration, queue browsing 167

Events
 sample 215
events
 notify undelivered 168
events, session 107
Exceptions
 GeneralException (ActiveX/COM) 263
 GeneralExceptionCode (ActiveX/COM) 263
 handling in the ActiveX/COM control 236
 handling on the connection 116
expiration 125, 140, 142, 170
 QoS level 36
Explorer
 checking default queues 45
 Dead Message Queue 66
 setting properties to preserve if undelivered 64
 startup 45

F

failover
 checking settings 98
 implementing 97
filters 145
flow control 109
 disabling 110

G

getPropertyNames 129
global
 queue setting 220
global queues 186

H

header fields 123
 default values 126
hierarchical name spaces 201
 as message filters 196
 sample application 77
host 43
hostname 96

I

IBrokerManagerListener 214
identifier 89
identifiers (ActiveX/COM) 233
indoubt messages 177
instanceof 51
integrity 34

J

JMS provider 26
JMSX properties 128
JNDI
 lookup of Destinations 105
 lookup of Topics 138, 191
 managing administered objects 268
JRE
 for ActiveX/COM 226
 installed 25
JVM 25
 identifying 21

L

latency 137
lazy acknowledgement 101
LDAP
 managing administered objects 269
Linux
 security database 86
 starting the SonicMQ Explorer 45
listeners 143, 163
 ActiveX/COM application 234
load balancing
 checking settings 98
 implementing 97
local
 queue setting 220
localhost 43
loop test 78

M

- Management API
 - Events sample 215
 - ShowSetup sample 221
 - Shutdown sample 223
- Map message
 - enhancing the sample 81
 - sample 49
- message
 - undelivered 177
- message ordering 136
 - PTP 161
 - Pub/Sub 198
- message reliability 136
 - PTP 161
 - Pub/Sub 199
- message selector 145
 - on QueueBrowser 167
 - sample 75, 76
- message server
 - definition 26
 - failure, handling 116
 - management of destination parameters 142
 - management of topic hierarchies 202
 - refresh settings 65
 - shutdown from an application 223
 - starting 43
- message traffic
 - Pub/Sub 54
- message types 103, 120
- Message_free (ActiveX/COM) 263
- MessageID 123
- method overloading
 - handling in ActiveX/COM 240

N

- name spaces 77, 201
- network failure 116
- noLocal 195, 196
- notification
 - when undelivered 172
- notify undelivered 128
- NoWait 144, 164

- null
 - in comparison tests 149
 - in topic naming 204

O

- object model 29
- one-to-many 27
- one-to-one 27

P

- password
 - prompt before server shutdown 223
- persistence
 - message delivery mode 123
 - on the message server 142
 - QoS options 34
- ping interval 114
- Point-to-Point 27
- port 43, 96
- prefetch
 - count 165
 - threshold 165
- preserve undelivered 128
- priority
 - default value 126
 - header field 125
 - on the message server 142
 - publish parameter 193
 - QoS level 35
- privacy 34
- producer 31, 138
- properties 127
 - dead message 68
- propertyExists 129
- protocol 96
- PTP 27
- Pub/Sub 27
- publish 126
 - method 193
- Publish and Subscribe 27
- publisher 138, 192

Q

QoS
 report in an application 222
 Quality of Protection
 listing 32
 Quality of Service
 listing 32
 sample
 durable subscription 60
 persistent storage 60
 reliable connection 60
 sample application 60
 queue
 browser 166
 browser sample 53
 dead messages 176
 default queues in database 45
 extents 220
 global 175
 listener 163
 remote 175
 set up 160
 show setup from an application 221
 size 220
 unbound 138

R

readFile 267
 reason codes 177
 receivers 144, 163
 multiple 163
 redelivered 35, 124
 redirect
 events 217
 reject
 events 216
 reliable connection 60
 remote queue 175
 replier 73, 74
 ReplyTo 124, 139
 request and reply 153
 QoS level 36, 37

 shared reply queues 187
 requestor 73, 74
 retry connection 99
 rollback 57
 definition 102
 routing
 problems causing non-delivery 177
 routing node
 message behaviors 175
 routing nodes 26
 routing table 178
 report in an application 222

S

samples
 Chat (ActiveX/COM) 227
 Chat (Pub/Sub) 46
 extended for common topics 79
 Dead Messages (PTP) 63
 DurableChat (Pub/Sub) 69
 extended for common topics 79
 Events (Management API) 215
 GlobalTalk 41
 HierarchicalChat (Pub/Sub) 77
 Map messages (PTP) 49
 extended for other data types 81
 MessageMonitor (Pub/Sub) 54
 QueueMonitor (PTP) 53
 ReliableChat (Pub/Sub) 60
 ReliableTalk (PTP) 61
 Request and Reply (PTP) 73
 Request and Reply (Pub/Sub) 74
 RoundTrip (PTP) 78, 80
 extended for various behaviors 80
 SelectorChat (Pub/Sub) 76
 SelectorTalk (PTP) 75
 SetupQueue (Management API) 221
 Shutdown (Management API) 223
 Talk (PTP) 47
 Transacted Messages (PTP) 56
 Transacted Messages (Pub/Sub) 57
 XMLChat (Pub/Sub) 48
 XMLMessage (PTP) 49

- XMLMessage (Pub/Sub) 49
 - extended with additional data 82
 - XMLTalk (PTP) 48
 - scripts
 - batch files 44
 - for compiling modified samples 44
 - for running samples 44
 - shell scripts 44
 - security
 - database 42
 - set up 86
 - enhanced samples 85
 - in topic name spaces 201
 - permission for publisher 192
 - selector string 75, 76
 - send 126
 - serialized Java objects 267
 - session 89
 - definition 30, 100
 - multiple 100
 - objects 103
 - SessionID (ActiveX/COM) 234
 - Queue_get 262
 - QueueBrowser_get 262
 - QueueReceiver_get 262
 - QueueSender_get 262
 - Topic_get 262
 - TopicSubscriber_get 262
 - setDocument 132
 - ShowSetup
 - sample 221
 - Shutdown
 - sample 223
 - single-message acknowledgement 101
 - SonicMQ Explorer
 - checking default queues 45
 - creating a publisher 192
 - creating a subscriber 194
 - message properties 129
 - publishing a message 194
 - starting 45
 - SQL 75, 76
 - SQL92 145
 - starting a connection 107
 - starting the message server 43
 - stopping a connection 107
 - subscriber
 - definition 195
 - durable 196
 - subscription name 91
 - support, technical 21
 - synchronous 143, 163
 - syntax
 - message selector string 146
 - notations used in this manual 17
 - SonicMQ ActiveX/COM methods 239
 - topic names 202
 - system queues 66
 - system topics 203
- ## T
- TCP_RESET 116
 - technical support 21
 - template characters 203, 206
 - topics 137, 191
 - temporary destination 72, 154, 186
 - Thread.sleep 99
 - threshold, prefetch 165
 - timeout 144, 164
 - timestamp 123
 - undelivered 128
 - time-to-live
 - default value 126
 - DurableChat sample 71
 - message property 128
 - on the message server 142
 - publish parameter 193
 - topic
 - common in samples 79
 - definition 191
 - hierarchical name spaces 137, 191
 - unbound 138, 192
 - topic hierarchy 201
 - TopicRequestor
 - listing of code 155
 - transacted
 - session parameter 100
 - transacted session
 - definition 102
 - type 124, 139
 - typographical conventions 17

U

- unbound 138, 192
- undelivered 177
 - events 217
 - notify 37, 64, 128
 - preserve 37, 64, 128
 - reason codes 128, 171, 177
 - timestamp 128
- UNIX
 - security database 86
 - shell scripts 44
 - starting the message server 43
 - starting the SonicMQ Explorer 45
- unsubscribe 25, 197
- URL 96
- user name 90
- users
 - report in an application 222

V

- valueOf 131
- Visual Basic 227

W

- wildcards 78
- Windows
 - security database 86
 - starting the message server 43
 - starting the SonicMQ Explorer 45

X

- XML message
 - ActiveX/COM application 235
 - create method 106
 - create through ActiveX/COM 262
 - creator syntax 121
 - enhanced sample 82
 - getDocument 133
 - sample application 48
- XML parser 48, 132

